

---

# **Abjad API**

***Release 2.13***

**Trevor Bača, Josiah Wolf Oberholtzer, Víctor Adán**

October 23, 2013



# CONTENTS

<b>I</b>	<b>Core composition packages</b>	<b>1</b>
<b>1</b>	<b>chordtools</b>	<b>3</b>
1.1	Concrete classes	3
1.1.1	chordtools.Chord	3
<b>2</b>	<b>componenttools</b>	<b>11</b>
2.1	Abstract classes	11
2.1.1	componenttools.Component	11
<b>3</b>	<b>containertools</b>	<b>13</b>
3.1	Concrete classes	13
3.1.1	containertools.Cluster	13
3.1.2	containertools.Container	19
3.1.3	containertools.FixedDurationContainer	25
3.1.4	containertools.GraceContainer	31
<b>4</b>	<b>contexttools</b>	<b>39</b>
4.1	Concrete classes	39
4.1.1	contexttools.ClefMark	39
4.1.2	contexttools.ClefMarkInventory	43
4.1.3	contexttools.Context	47
4.1.4	contexttools.ContextMark	53
4.1.5	contexttools.DynamicMark	55
4.1.6	contexttools.KeySignatureMark	58
4.1.7	contexttools.StaffChangeMark	61
4.1.8	contexttools.TempoMark	64
4.1.9	contexttools.TempoMarkInventory	68
4.1.10	contexttools.TimeSignatureMark	72
<b>5</b>	<b>durationtools</b>	<b>77</b>
5.1	Concrete classes	77
5.1.1	durationtools.Duration	77
5.1.2	durationtools.Multiplier	87
5.1.3	durationtools.Offset	96
<b>6</b>	<b>instrumenttools</b>	<b>107</b>
6.1	Concrete classes	107
6.1.1	instrumenttools.Accordion	107
6.1.2	instrumenttools.AltoFlute	113
6.1.3	instrumenttools.AltoSaxophone	119
6.1.4	instrumenttools.AltoTrombone	124
6.1.5	instrumenttools.BFlatClarinet	130
6.1.6	instrumenttools.BaritoneSaxophone	133
6.1.7	instrumenttools.BaritoneVoice	136

6.1.8	instrumenttools.BassClarinet	139
6.1.9	instrumenttools.BassFlute	142
6.1.10	instrumenttools.BassSaxophone	145
6.1.11	instrumenttools.BassTrombone	148
6.1.12	instrumenttools.BassVoice	151
6.1.13	instrumenttools.Bassoon	154
6.1.14	instrumenttools.Cello	157
6.1.15	instrumenttools.ClarinetInA	160
6.1.16	instrumenttools.Contrabass	163
6.1.17	instrumenttools.ContrabassClarinet	166
6.1.18	instrumenttools.ContrabassFlute	169
6.1.19	instrumenttools.ContrabassSaxophone	172
6.1.20	instrumenttools.Contrabassoon	175
6.1.21	instrumenttools.ContraltoVoice	178
6.1.22	instrumenttools.EFlatClarinet	181
6.1.23	instrumenttools.EnglishHorn	184
6.1.24	instrumenttools.Flute	187
6.1.25	instrumenttools.FrenchHorn	190
6.1.26	instrumenttools.Glockenspiel	193
6.1.27	instrumenttools.Guitar	196
6.1.28	instrumenttools.Harp	199
6.1.29	instrumenttools.Harpsichord	202
6.1.30	instrumenttools.Instrument	205
6.1.31	instrumenttools.InstrumentInventory	208
6.1.32	instrumenttools.Marimba	212
6.1.33	instrumenttools.MezzoSopranoVoice	215
6.1.34	instrumenttools.Oboe	218
6.1.35	instrumenttools.Piano	221
6.1.36	instrumenttools.Piccolo	224
6.1.37	instrumenttools.SopraninoSaxophone	227
6.1.38	instrumenttools.SopranoSaxophone	230
6.1.39	instrumenttools.SopranoVoice	233
6.1.40	instrumenttools.TenorSaxophone	236
6.1.41	instrumenttools.TenorTrombone	239
6.1.42	instrumenttools.TenorVoice	242
6.1.43	instrumenttools.Trumpet	245
6.1.44	instrumenttools.Tuba	248
6.1.45	instrumenttools.UntunedPercussion	251
6.1.46	instrumenttools.Vibraphone	254
6.1.47	instrumenttools.Viola	257
6.1.48	instrumenttools.Violin	260
6.1.49	instrumenttools.WoodwindFingering	263
6.1.50	instrumenttools.Xylophone	267
6.2	Functions	269
6.2.1	instrumenttools.default_instrument_name_to_instrument_class	269
6.2.2	instrumenttools.iterate_out_of_range_notes_and_chords	270
6.2.3	instrumenttools.notes_and_chords_are_in_range	270
6.2.4	instrumenttools.notes_and_chords_are_on_expected_clefs	270
6.2.5	instrumenttools.transpose_from_sounding_pitch_to_written_pitch	271
6.2.6	instrumenttools.transpose_from_written_pitch_to_sounding_pitch	271
<b>7</b>	<b>iotools</b>	<b>273</b>
7.1	Concrete classes	273
7.1.1	iotools.RedirectedStreams	273
7.2	Functions	274
7.2.1	iotools.clear_terminal	274
7.2.2	iotools.count_function_calls	274
7.2.3	iotools.f	275



7.2.4	iotools.get_last_output_file_name . . . . .	275
7.2.5	iotools.get_next_output_file_name . . . . .	275
7.2.6	iotools.graph . . . . .	275
7.2.7	iotools.insert_expr_into_lilypond_file . . . . .	276
7.2.8	iotools.log . . . . .	276
7.2.9	iotools.log_render_lilypond_input . . . . .	276
7.2.10	iotools.ly . . . . .	277
7.2.11	iotools.open_file . . . . .	277
7.2.12	iotools.p . . . . .	277
7.2.13	iotools.pdf . . . . .	277
7.2.14	iotools.play . . . . .	278
7.2.15	iotools.plot . . . . .	278
7.2.16	iotools.profile_expr . . . . .	278
7.2.17	iotools.redo . . . . .	279
7.2.18	iotools.run_abjad . . . . .	279
7.2.19	iotools.run_lilypond . . . . .	279
7.2.20	iotools.save_last_ly_as . . . . .	279
7.2.21	iotools.save_last_pdf_as . . . . .	279
7.2.22	iotools.show . . . . .	280
7.2.23	iotools.spawn_subprocess . . . . .	280
7.2.24	iotools.verify_output_directory . . . . .	280
7.2.25	iotools.warn_almost_full . . . . .	280
7.2.26	iotools.which . . . . .	281
7.2.27	iotools.write_expr_to_ly . . . . .	281
7.2.28	iotools.write_expr_to_pdf . . . . .	281
7.2.29	iotools.z . . . . .	281

## 8 iterationtools 283

8.1	Functions . . . . .	283
8.1.1	iterationtools.iterate_chords_in_expr . . . . .	283
8.1.2	iterationtools.iterate_components_and_grace_containers_in_expr . . . . .	283
8.1.3	iterationtools.iterate_components_depth_first . . . . .	284
8.1.4	iterationtools.iterate_components_in_expr . . . . .	284
8.1.5	iterationtools.iterate_containers_in_expr . . . . .	284
8.1.6	iterationtools.iterate_contexts_in_expr . . . . .	285
8.1.7	iterationtools.iterate_leaf_pairs_in_expr . . . . .	285
8.1.8	iterationtools.iterate_leaves_in_expr . . . . .	285
8.1.9	iterationtools.iterate_logical_voice_from_component . . . . .	287
8.1.10	iterationtools.iterate_logical_voice_in_expr . . . . .	288
8.1.11	iterationtools.iterate_measures_in_expr . . . . .	289
8.1.12	iterationtools.iterate_nontrivial_tie_chains_in_expr . . . . .	289
8.1.13	iterationtools.iterate_notes_and_chords_in_expr . . . . .	290
8.1.14	iterationtools.iterate_notes_in_expr . . . . .	290
8.1.15	iterationtools.iterate_pitched_tie_chains_in_expr . . . . .	292
8.1.16	iterationtools.iterate_rests_in_expr . . . . .	292
8.1.17	iterationtools.iterate_runs_in_expr . . . . .	292
8.1.18	iterationtools.iterate_scores_in_expr . . . . .	293
8.1.19	iterationtools.iterate_semantic_voices_in_expr . . . . .	293
8.1.20	iterationtools.iterate_skips_in_expr . . . . .	294
8.1.21	iterationtools.iterate_staves_in_expr . . . . .	294
8.1.22	iterationtools.iterate_tie_chains_in_expr . . . . .	294
8.1.23	iterationtools.iterate_timeline_from_component . . . . .	295
8.1.24	iterationtools.iterate_timeline_in_expr . . . . .	295
8.1.25	iterationtools.iterate_topmost_tie_chains_and_components_in_expr . . . . .	296
8.1.26	iterationtools.iterate_tuplets_in_expr . . . . .	297
8.1.27	iterationtools.iterate_vertical_moments_in_expr . . . . .	297
8.1.28	iterationtools.iterate_voices_in_expr . . . . .	298

<b>9</b>	<b>labeltools</b>	<b>299</b>
9.1	Functions	299
9.1.1	labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map	299
9.1.2	labeltools.color_contents_of_container	299
9.1.3	labeltools.color_leaf	300
9.1.4	labeltools.color_leaves_in_expr	300
9.1.5	labeltools.color_measure	301
9.1.6	labeltools.color_measures_with_non_power_of_two_denominators_in_expr	301
9.1.7	labeltools.color_note_head_by_numbered_pitch_class_color_map	302
9.1.8	labeltools.label_leaves_in_expr_with_leaf_depth	302
9.1.9	labeltools.label_leaves_in_expr_with_leaf_duration	303
9.1.10	labeltools.label_leaves_in_expr_with_leaf_durations	303
9.1.11	labeltools.label_leaves_in_expr_with_leaf_indices	304
9.1.12	labeltools.label_leaves_in_expr_with_leaf_numbers	304
9.1.13	labeltools.label_leaves_in_expr_with_named_interval_classes	305
9.1.14	labeltools.label_leaves_in_expr_with_named_intervals	305
9.1.15	labeltools.label_leaves_in_expr_with_numbered_interval_classes	305
9.1.16	labeltools.label_leaves_in_expr_with_numbered_intervals	306
9.1.17	labeltools.label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes	306
9.1.18	labeltools.label_leaves_in_expr_with_pitch_class_numbers	306
9.1.19	labeltools.label_leaves_in_expr_with_pitch_numbers	307
9.1.20	labeltools.label_leaves_in_expr_with_tuplet_depth	307
9.1.21	labeltools.label_leaves_in_expr_with_written_leaf_duration	308
9.1.22	labeltools.label_notes_in_expr_with_note_indices	308
9.1.23	labeltools.label_tie_chains_in_expr_with_tie_chain_duration	308
9.1.24	labeltools.label_tie_chains_in_expr_with_tie_chain_durations	309
9.1.25	labeltools.label_tie_chains_in_expr_with_written_tie_chain_duration	309
9.1.26	labeltools.label_vertical_moments_in_expr_with_interval_class_vectors	309
9.1.27	labeltools.label_vertical_moments_in_expr_with_named_intervals	310
9.1.28	labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes	310
9.1.29	labeltools.label_vertical_moments_in_expr_with_numbered_intervals	311
9.1.30	labeltools.label_vertical_moments_in_expr_with_numbered_pitch_classes	311
9.1.31	labeltools.label_vertical_moments_in_expr_with_pitch_numbers	312
9.1.32	labeltools.remove_markup_from_leaves_in_expr	312
<b>10</b>	<b>layouttools</b>	<b>315</b>
10.1	Concrete classes	315
10.1.1	layouttools.SpacingIndication	315
10.2	Functions	316
10.2.1	layouttools.make_spacing_vector	316
10.2.2	layouttools.set_line_breaks_by_line_duration	317
10.2.3	layouttools.set_line_breaks_cyclically_by_line_duration_ge	317
10.2.4	layouttools.set_line_breaks_cyclically_by_line_duration_in_seconds_ge	318
<b>11</b>	<b>leaftools</b>	<b>321</b>
11.1	Abstract classes	321
11.1.1	leaftools.Leaf	321
11.2	Functions	323
11.2.1	leaftools.make_leaves	323
11.2.2	leaftools.make_leaves_from_talea	326
11.2.3	leaftools.make_tied_leaf	327
<b>12</b>	<b>lilypondfiletools</b>	<b>329</b>
12.1	Abstract classes	329
12.1.1	lilypondfiletools.AttributedBlock	329
12.1.2	lilypondfiletools.NonattributedBlock	332
12.2	Concrete classes	334
12.2.1	lilypondfiletools.AbjadRevisionToken	334
12.2.2	lilypondfiletools.BookBlock	335

12.2.3	<code>lilypondfiletools.BookpartBlock</code>	338
12.2.4	<code>lilypondfiletools.ContextBlock</code>	340
12.2.5	<code>lilypondfiletools.DateToken</code>	343
12.2.6	<code>lilypondfiletools.HeaderBlock</code>	344
12.2.7	<code>lilypondfiletools.LayoutBlock</code>	347
12.2.8	<code>lilypondfiletools.LilyPondDimension</code>	350
12.2.9	<code>lilypondfiletools.LilyPondFile</code>	351
12.2.10	<code>lilypondfiletools.LilyPondLanguageToken</code>	354
12.2.11	<code>lilypondfiletools.LilyPondVersionToken</code>	355
12.2.12	<code>lilypondfiletools.MIDIBlock</code>	356
12.2.13	<code>lilypondfiletools.PaperBlock</code>	359
12.2.14	<code>lilypondfiletools.ScoreBlock</code>	361
12.3	Functions	364
12.3.1	<code>lilypondfiletools.make_basic_lilypond_file</code>	364
12.3.2	<code>lilypondfiletools.make_floating_time_signature_lilypond_file</code>	364
12.3.3	<code>lilypondfiletools.make_time_signature_context_block</code>	364
<b>13</b>	<b>marktools</b>	<b>365</b>
13.1	Abstract classes	365
13.1.1	<code>marktools.DirectedMark</code>	365
13.2	Concrete classes	367
13.2.1	<code>marktools.Annotation</code>	367
13.2.2	<code>marktools.Articulation</code>	369
13.2.3	<code>marktools.BarLine</code>	372
13.2.4	<code>marktools.BendAfter</code>	375
13.2.5	<code>marktools.LilyPondCommandMark</code>	377
13.2.6	<code>marktools.LilyPondComment</code>	379
13.2.7	<code>marktools.Mark</code>	382
13.2.8	<code>marktools.StemTremolo</code>	383
<b>14</b>	<b>markuptools</b>	<b>387</b>
14.1	Concrete classes	387
14.1.1	<code>markuptools.Markup</code>	387
14.1.2	<code>markuptools.MarkupCommand</code>	390
14.1.3	<code>markuptools.MarkupInventory</code>	392
14.1.4	<code>markuptools.MusicGlyph</code>	396
14.2	Functions	397
14.2.1	<code>markuptools.combine_markup_commands</code>	397
14.2.2	<code>markuptools.make_big_centered_page_number_markup</code>	397
14.2.3	<code>markuptools.make_blank_line_markup</code>	398
14.2.4	<code>markuptools.make_centered_title_markup</code>	398
14.2.5	<code>markuptools.make_vertically_adjusted_composer_markup</code>	398
<b>15</b>	<b>mathtools</b>	<b>401</b>
15.1	Concrete classes	401
15.1.1	<code>mathtools.BoundedObject</code>	401
15.1.2	<code>mathtools.Infinity</code>	403
15.1.3	<code>mathtools.NegativeInfinity</code>	404
15.1.4	<code>mathtools.NonreducedFraction</code>	406
15.1.5	<code>mathtools.NonreducedRatio</code>	413
15.1.6	<code>mathtools.Ratio</code>	415
15.2	Functions	417
15.2.1	<code>mathtools.are_relatively_prime</code>	417
15.2.2	<code>mathtools.arithmetic_mean</code>	417
15.2.3	<code>mathtools.binomial_coefficient</code>	417
15.2.4	<code>mathtools.cumulative_products</code>	418
15.2.5	<code>mathtools.cumulative_signed_weights</code>	418
15.2.6	<code>mathtools.cumulative_sums</code>	418
15.2.7	<code>mathtools.cumulative_sums_zero</code>	418

15.2.8	mathtools.cumulative_sums_zero_pairwise	419
15.2.9	mathtools.difference_series	419
15.2.10	mathtools.divide_number_by_ratio	419
15.2.11	mathtools.divisors	419
15.2.12	mathtools.factors	420
15.2.13	mathtools.fraction_to_proper_fraction	420
15.2.14	mathtools.get_shared_numeric_sign	421
15.2.15	mathtools.greatest_common_divisor	421
15.2.16	mathtools.greatest_multiple_less_equal	421
15.2.17	mathtools.greatest_power_of_two_less_equal	422
15.2.18	mathtools.integer_equivalent_number_to_integer	422
15.2.19	mathtools.integer_to_base_k_tuple	423
15.2.20	mathtools.integer_to_binary_string	423
15.2.21	mathtools.interpolate_cosine	423
15.2.22	mathtools.interpolate_divide	423
15.2.23	mathtools.interpolate_divide_multiple	424
15.2.24	mathtools.interpolate_exponential	424
15.2.25	mathtools.interpolate_linear	424
15.2.26	mathtools.interval_string_to_pair_and_indicators	425
15.2.27	mathtools.is_assignable_integer	425
15.2.28	mathtools.is_dotted_integer	425
15.2.29	mathtools.is_integer_equivalent_expr	426
15.2.30	mathtools.is_integer_equivalent_number	426
15.2.31	mathtools.is_negative_integer	426
15.2.32	mathtools.is_nonnegative_integer	427
15.2.33	mathtools.is_nonnegative_integer_equivalent_number	427
15.2.34	mathtools.is_nonnegative_integer_power_of_two	427
15.2.35	mathtools.is_positive_integer	427
15.2.36	mathtools.is_positive_integer_equivalent_number	428
15.2.37	mathtools.is_positive_integer_power_of_two	428
15.2.38	mathtools.least_common_multiple	428
15.2.39	mathtools.least_multiple_greater_equal	428
15.2.40	mathtools.least_power_of_two_greater_equal	429
15.2.41	mathtools.next_integer_partition	430
15.2.42	mathtools.partition_integer_by_ratio	430
15.2.43	mathtools.partition_integer_into_canonic_parts	430
15.2.44	mathtools.partition_integer_into_halves	431
15.2.45	mathtools.partition_integer_into_parts_less_than_double	432
15.2.46	mathtools.partition_integer_into_units	432
15.2.47	mathtools.remove_powers_of_two	433
15.2.48	mathtools.sign	433
15.2.49	mathtools.weight	433
15.2.50	mathtools.yield_all_compositions_of_integer	433
15.2.51	mathtools.yield_all_partitions_of_integer	434
15.2.52	mathtools.yield_nonreduced_fractions	434
<b>16</b>	<b>measuretools</b>	<b>437</b>
16.1	Concrete classes	437
16.1.1	measuretools.Measure	437
16.2	Functions	445
16.2.1	measuretools.append_spacer_skip_to_underfull_measure	445
16.2.2	measuretools.append_spacer_skips_to_underfull_measures_in_expr	445
16.2.3	measuretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr	446
16.2.4	measuretools.extend_measures_in_expr_and_apply_full_measure_tuplets	446
16.2.5	measuretools.fill_measures_in_expr_with_full_measure_spacer_skips	447
16.2.6	measuretools.fill_measures_in_expr_with_minimal_number_of_notes	447
16.2.7	measuretools.fill_measures_in_expr_with_repeated_notes	447
16.2.8	measuretools.fill_measures_in_expr_with_time_signature_denominator_notes	447

16.2.9	measuretools.get_measure_that_starts_with_container	447
16.2.10	measuretools.get_measure_that_stops_with_container	448
16.2.11	measuretools.get_next_measure_from_component	448
16.2.12	measuretools.get_one_indexed_measure_number_in_expr	448
16.2.13	measuretools.get_previous_measure_from_component	449
16.2.14	measuretools.make_measures_with_full_measure_spacer_skips	449
16.2.15	measuretools.move_full_measure_tuplet_prolation_to_measure_time_signature	449
16.2.16	measuretools.move_measure_prolation_to_full_measure_tuplet	450
16.2.17	measuretools.replace_contents_of_measures_in_expr	450
16.2.18	measuretools.scale_measure_denominator_and_adjust_measure_contents	450
16.2.19	measuretools.set_always_format_time_signature_of_measures_in_expr	451
16.2.20	measuretools.set_measure_denominator_and_adjust_numerator	451
<b>17</b>	<b>mutationtools</b>	<b>453</b>
17.1	Concrete classes	453
17.1.1	mutationtools.AttributeInspectionAgent	453
17.1.2	mutationtools.ScoreMutationAgent	457
17.2	Functions	467
17.2.1	mutationtools.mutate	467
<b>18</b>	<b>notetools</b>	<b>469</b>
18.1	Concrete classes	469
18.1.1	notetools.Harmonic	469
18.1.2	notetools.NaturalHarmonic	470
18.1.3	notetools.Note	474
18.1.4	notetools.NoteHead	477
18.2	Functions	479
18.2.1	notetools.make_accelerating_notes_with_lilypond_multipliers	479
18.2.2	notetools.make_notes	480
18.2.3	notetools.make_notes_with_multiplied_durations	480
18.2.4	notetools.make_percussion_note	481
18.2.5	notetools.make_quarter_notes_with_lilypond_duration_multiplier	481
18.2.6	notetools.make_repeated_notes	482
18.2.7	notetools.make_repeated_notes_from_time_signature	482
18.2.8	notetools.make_repeated_notes_from_time_signatures	482
18.2.9	notetools.make_repeated_notes_with_shorter_notes_at_end	482
<b>19</b>	<b>pitcharraytools</b>	<b>485</b>
19.1	Concrete classes	485
19.1.1	pitcharraytools.PitchArray	485
19.1.2	pitcharraytools.PitchArrayCell	489
19.1.3	pitcharraytools.PitchArrayColumn	491
19.1.4	pitcharraytools.PitchArrayInventory	493
19.1.5	pitcharraytools.PitchArrayRow	497
<b>20</b>	<b>pitchtools</b>	<b>501</b>
20.1	Abstract classes	501
20.1.1	pitchtools.Interval	501
20.1.2	pitchtools.IntervalClass	503
20.1.3	pitchtools.Pitch	504
20.1.4	pitchtools.PitchClass	507
20.1.5	pitchtools.Segment	510
20.1.6	pitchtools.Set	512
20.1.7	pitchtools.Vector	514
20.2	Concrete classes	516
20.2.1	pitchtools.Accidental	516
20.2.2	pitchtools.IntervalClassSegment	519
20.2.3	pitchtools.IntervalClassSet	522
20.2.4	pitchtools.IntervalClassVector	525

20.2.5	pitchtools.IntervalSegment	528
20.2.6	pitchtools.IntervalSet	531
20.2.7	pitchtools.IntervalVector	534
20.2.8	pitchtools.NamedInterval	537
20.2.9	pitchtools.NamedIntervalClass	539
20.2.10	pitchtools.NamedInversionEquivalentIntervalClass	541
20.2.11	pitchtools.NamedPitch	542
20.2.12	pitchtools.NamedPitchClass	547
20.2.13	pitchtools.NumberedInterval	551
20.2.14	pitchtools.NumberedIntervalClass	553
20.2.15	pitchtools.NumberedInversionEquivalentIntervalClass	555
20.2.16	pitchtools.NumberedPitch	556
20.2.17	pitchtools.NumberedPitchClass	561
20.2.18	pitchtools.NumberedPitchClassColorMap	565
20.2.19	pitchtools.Octave	566
20.2.20	pitchtools.OctaveTranspositionMapping	569
20.2.21	pitchtools.OctaveTranspositionMappingComponent	573
20.2.22	pitchtools.OctaveTranspositionMappingInventory	575
20.2.23	pitchtools.PitchClassSegment	579
20.2.24	pitchtools.PitchClassSet	583
20.2.25	pitchtools.PitchClassVector	587
20.2.26	pitchtools.PitchRange	589
20.2.27	pitchtools.PitchRangeInventory	592
20.2.28	pitchtools.PitchSegment	596
20.2.29	pitchtools.PitchSet	600
20.2.30	pitchtools.PitchVector	603
20.2.31	pitchtools.TwelveToneRow	605
20.3	Functions	608
20.3.1	pitchtools.apply_accidental_to_named_pitch	608
20.3.2	pitchtools.clef_and_staff_position_number_to_named_pitch	609
20.3.3	pitchtools.contains_subsegment	609
20.3.4	pitchtools.get_named_pitch_from_pitch_carrier	609
20.3.5	pitchtools.get_numbered_pitch_class_from_pitch_carrier	610
20.3.6	pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list	610
20.3.7	pitchtools.instantiate_pitch_and_interval_test_collection	611
20.3.8	pitchtools.inventory_aggregate_subsets	611
20.3.9	pitchtools.iterate_named_pitch_pairs_in_expr	612
20.3.10	pitchtools.list_named_pitches_in_expr	612
20.3.11	pitchtools.list_numbered_interval_numbers_pairwise	613
20.3.12	pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise	613
20.3.13	pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range	614
20.3.14	pitchtools.list_ordered_named_pitch_pairs_from_expr_1_to_expr_2	615
20.3.15	pitchtools.list_pitch_numbers_in_expr	615
20.3.16	pitchtools.list_unordered_named_pitch_pairs_in_expr	615
20.3.17	pitchtools.make_n_middle_c_centered_pitches	616
20.3.18	pitchtools.named_pitch_and_clef_to_staff_position_number	616
20.3.19	pitchtools.numbered_inversion_equivalent_interval_class_dictionary	616
20.3.20	pitchtools.permute_named_pitch_carrier_list_by_twelve_tone_row	617
20.3.21	pitchtools.register_pitch_class_numbers_by_pitch_number_aggregate	617
20.3.22	pitchtools.set_written_pitch_of_pitched_components_in_expr	617
20.3.23	pitchtools.sort_named_pitch_carriers_in_expr	617
20.3.24	pitchtools.spell_numbered_interval_number	617
20.3.25	pitchtools.spell_pitch_number	618
20.3.26	pitchtools.suggest_clef_for_named_pitches	618
20.3.27	pitchtools.transpose_named_pitch_by_numbered_interval_and_respell	618
20.3.28	pitchtools.transpose_pitch_carrier_by_interval	618
20.3.29	pitchtools.transpose_pitch_class_number_to_pitch_number_neighbor	619
20.3.30	pitchtools.transpose_pitch_expr_into_pitch_range	619

20.3.31	pitchtools.transpose_pitch_number_by_octave_transposition_mapping . . . . .	619
<b>21</b>	<b>quantizationtools</b>	<b>621</b>
21.1	Abstract classes . . . . .	621
21.1.1	quantizationtools.AttackPointOptimizer . . . . .	621
21.1.2	quantizationtools.GraceHandler . . . . .	623
21.1.3	quantizationtools.Heuristic . . . . .	624
21.1.4	quantizationtools.JobHandler . . . . .	625
21.1.5	quantizationtools.QEvent . . . . .	626
21.1.6	quantizationtools.QSchema . . . . .	628
21.1.7	quantizationtools.QSchemaItem . . . . .	629
21.1.8	quantizationtools.QTarget . . . . .	631
21.1.9	quantizationtools.SearchTree . . . . .	632
21.2	Concrete classes . . . . .	634
21.2.1	quantizationtools.BeatwiseQSchema . . . . .	634
21.2.2	quantizationtools.BeatwiseQSchemaItem . . . . .	637
21.2.3	quantizationtools.BeatwiseQTarget . . . . .	639
21.2.4	quantizationtools.CollapsingGraceHandler . . . . .	640
21.2.5	quantizationtools.ConcatenatingGraceHandler . . . . .	641
21.2.6	quantizationtools.DiscardingGraceHandler . . . . .	642
21.2.7	quantizationtools.DistanceHeuristic . . . . .	643
21.2.8	quantizationtools.MeasurewiseAttackPointOptimizer . . . . .	644
21.2.9	quantizationtools.MeasurewiseQSchema . . . . .	645
21.2.10	quantizationtools.MeasurewiseQSchemaItem . . . . .	649
21.2.11	quantizationtools.MeasurewiseQTarget . . . . .	651
21.2.12	quantizationtools.NaiveAttackPointOptimizer . . . . .	652
21.2.13	quantizationtools.NullAttackPointOptimizer . . . . .	653
21.2.14	quantizationtools.ParallelJobHandler . . . . .	654
21.2.15	quantizationtools.ParallelJobHandlerWorker . . . . .	655
21.2.16	quantizationtools.PitchedQEvent . . . . .	657
21.2.17	quantizationtools.QEventProxy . . . . .	658
21.2.18	quantizationtools.QEventSequence . . . . .	659
21.2.19	quantizationtools.QGrid . . . . .	665
21.2.20	quantizationtools.QGridContainer . . . . .	668
21.2.21	quantizationtools.QGridLeaf . . . . .	680
21.2.22	quantizationtools.QTargetBeat . . . . .	685
21.2.23	quantizationtools.QTargetMeasure . . . . .	687
21.2.24	quantizationtools.QuantizationJob . . . . .	690
21.2.25	quantizationtools.Quantizer . . . . .	693
21.2.26	quantizationtools.SerialJobHandler . . . . .	696
21.2.27	quantizationtools.SilentQEvent . . . . .	697
21.2.28	quantizationtools.TerminalQEvent . . . . .	699
21.2.29	quantizationtools.UnweightedSearchTree . . . . .	700
21.2.30	quantizationtools.WeightedSearchTree . . . . .	702
21.3	Functions . . . . .	704
21.3.1	quantizationtools.make_test_time_segments . . . . .	704
<b>22</b>	<b>resttools</b>	<b>705</b>
22.1	Concrete classes . . . . .	705
22.1.1	resttools.MultimeasureRest . . . . .	705
22.1.2	resttools.Rest . . . . .	708
22.2	Functions . . . . .	710
22.2.1	resttools.make_multimeasure_rests . . . . .	710
22.2.2	resttools.make_repeated_rests_from_time_signatures . . . . .	710
22.2.3	resttools.make_rests . . . . .	710
<b>23</b>	<b>rhythmmakertools</b>	<b>713</b>
23.1	Abstract classes . . . . .	713
23.1.1	rhythmmakertools.BurnishedRhythmMaker . . . . .	713



23.1.2	rhythmmakertools.DivisionIncisedRhythmMaker	716
23.1.3	rhythmmakertools.IncisedRhythmMaker	719
23.1.4	rhythmmakertools.OutputIncisedRhythmMaker	721
23.1.5	rhythmmakertools.RhythmMaker	723
23.2	Concrete classes	725
23.2.1	rhythmmakertools.DivisionBurnishedTaleaRhythmMaker	725
23.2.2	rhythmmakertools.DivisionIncisedNoteRhythmMaker	730
23.2.3	rhythmmakertools.DivisionIncisedRestRhythmMaker	735
23.2.4	rhythmmakertools.EqualDivisionRhythmMaker	739
23.2.5	rhythmmakertools.EvenRunRhythmMaker	742
23.2.6	rhythmmakertools.NoteRhythmMaker	745
23.2.7	rhythmmakertools.OutputBurnishedTaleaRhythmMaker	748
23.2.8	rhythmmakertools.OutputIncisedNoteRhythmMaker	752
23.2.9	rhythmmakertools.OutputIncisedRestRhythmMaker	756
23.2.10	rhythmmakertools.RestRhythmMaker	760
23.2.11	rhythmmakertools.SkipRhythmMaker	763
23.2.12	rhythmmakertools.TaleaRhythmMaker	766
23.2.13	rhythmmakertools.TupletMonadRhythmMaker	769
<b>24</b>	<b>rhythmtreetools</b>	<b>773</b>
24.1	Abstract classes	773
24.1.1	rhythmtreetools.RhythmTreeNode	773
24.2	Concrete classes	778
24.2.1	rhythmtreetools.RhythmTreeContainer	778
24.2.2	rhythmtreetools.RhythmTreeLeaf	791
24.2.3	rhythmtreetools.RhythmTreeParser	796
24.3	Functions	799
24.3.1	rhythmtreetools.parse_rtm_syntax	799
<b>25</b>	<b>schemetools</b>	<b>801</b>
25.1	Concrete classes	801
25.1.1	schemetools.Scheme	801
25.1.2	schemetools.SchemeAssociativeList	804
25.1.3	schemetools.SchemeColor	806
25.1.4	schemetools.SchemeMoment	808
25.1.5	schemetools.SchemePair	810
25.1.6	schemetools.SchemeVector	811
25.1.7	schemetools.SchemeVectorConstant	813
<b>26</b>	<b>scoretemplatetools</b>	<b>815</b>
26.1	Concrete classes	815
26.1.1	scoretemplatetools.GroupedRhythmicStavesScoreTemplate	815
26.1.2	scoretemplatetools.GroupedStavesScoreTemplate	817
26.1.3	scoretemplatetools.StringOrchestraScoreTemplate	818
26.1.4	scoretemplatetools.StringQuartetScoreTemplate	820
26.1.5	scoretemplatetools.TwoStaffPianoScoreTemplate	821
<b>27</b>	<b>scoretools</b>	<b>823</b>
27.1	Concrete classes	823
27.1.1	scoretools.GrandStaff	823
27.1.2	scoretools.InstrumentationSpecifier	830
27.1.3	scoretools.Performer	832
27.1.4	scoretools.PerformerInventory	838
27.1.5	scoretools.PianoStaff	842
27.1.6	scoretools.Score	849
27.1.7	scoretools.StaffGroup	857
27.2	Functions	863
27.2.1	scoretools.make_empty_piano_score	863
27.2.2	scoretools.make_piano_score_from_leaves	863



27.2.3	scoretools.make_piano_sketch_score_from_leaves	864
<b>28</b>	<b>selectiontools</b>	<b>865</b>
28.1	Concrete classes	865
28.1.1	selectiontools.ContiguousSelection	865
28.1.2	selectiontools.Descendants	867
28.1.3	selectiontools.Lineage	869
28.1.4	selectiontools.Parentage	871
28.1.5	selectiontools.Selection	875
28.1.6	selectiontools.SelectionInventory	877
28.1.7	selectiontools.SimultaneousSelection	881
28.1.8	selectiontools.SliceSelection	883
28.1.9	selectiontools.TieChain	885
28.1.10	selectiontools.VerticalMoment	889
28.2	Functions	891
28.2.1	selectiontools.select	891
<b>29</b>	<b>sequencetools</b>	<b>893</b>
29.1	Functions	893
29.1.1	sequencetools.all_are_assignable_integers	893
29.1.2	sequencetools.all_are_equal	893
29.1.3	sequencetools.all_are_integer_equivalent_exprs	893
29.1.4	sequencetools.all_are_integer_equivalent_numbers	894
29.1.5	sequencetools.all_are_nonnegative_integer_equivalent_numbers	894
29.1.6	sequencetools.all_are_nonnegative_integer_powers_of_two	894
29.1.7	sequencetools.all_are_nonnegative_integers	894
29.1.8	sequencetools.all_are_numbers	895
29.1.9	sequencetools.all_are_pairs	895
29.1.10	sequencetools.all_are_pairs_of_types	895
29.1.11	sequencetools.all_are_positive_integer_equivalent_numbers	896
29.1.12	sequencetools.all_are_positive_integers	896
29.1.13	sequencetools.all_are_unequal	896
29.1.14	sequencetools.count_length_two_runs_in_sequence	896
29.1.15	sequencetools.divide_sequence_elements_by_greatest_common_divisor	897
29.1.16	sequencetools.flatten_sequence	897
29.1.17	sequencetools.flatten_sequence_at_indices	897
29.1.18	sequencetools.get_indices_of_sequence_elements_equal_to_true	898
29.1.19	sequencetools.get_sequence_degree_of_rotational_symmetry	898
29.1.20	sequencetools.get_sequence_element_at_cyclic_index	898
29.1.21	sequencetools.get_sequence_elements_at_indices	899
29.1.22	sequencetools.get_sequence_elements_frequency_distribution	899
29.1.23	sequencetools.get_sequence_period_of_rotation	899
29.1.24	sequencetools.increase_sequence_elements_at_indices_by_addenda	899
29.1.25	sequencetools.increase_sequence_elements_cyclically_by_addenda	900
29.1.26	sequencetools.interlace_sequences	900
29.1.27	sequencetools.is_fraction_equivalent_pair	900
29.1.28	sequencetools.is_integer_equivalent_n_tuple	900
29.1.29	sequencetools.is_integer_equivalent_pair	901
29.1.30	sequencetools.is_integer_equivalent_singleton	901
29.1.31	sequencetools.is_integer_n_tuple	901
29.1.32	sequencetools.is_integer_pair	901
29.1.33	sequencetools.is_integer_singleton	902
29.1.34	sequencetools.is_monotonically_decreasing_sequence	902
29.1.35	sequencetools.is_monotonically_increasing_sequence	902
29.1.36	sequencetools.is_n_tuple	903
29.1.37	sequencetools.is_null_tuple	903
29.1.38	sequencetools.is_pair	903
29.1.39	sequencetools.is_permutation	904

29.1.40	sequencetools.is_repetition_free_sequence . . . . .	904
29.1.41	sequencetools.is_restricted_growth_function . . . . .	904
29.1.42	sequencetools.is_singleton . . . . .	905
29.1.43	sequencetools.is_strictly_decreasing_sequence . . . . .	905
29.1.44	sequencetools.is_strictly_increasing_sequence . . . . .	906
29.1.45	sequencetools.iterate_sequence_cyclically . . . . .	906
29.1.46	sequencetools.iterate_sequence_cyclically_from_start_to_stop . . . . .	907
29.1.47	sequencetools.iterate_sequence_forward_and_backward_nonoverlapping . . . . .	907
29.1.48	sequencetools.iterate_sequence_forward_and_backward_overlapping . . . . .	907
29.1.49	sequencetools.iterate_sequence_nwise_cyclic . . . . .	907
29.1.50	sequencetools.iterate_sequence_nwise_strict . . . . .	908
29.1.51	sequencetools.iterate_sequence_nwise_wrapped . . . . .	908
29.1.52	sequencetools.iterate_sequence_pairwise_cyclic . . . . .	908
29.1.53	sequencetools.iterate_sequence_pairwise_strict . . . . .	909
29.1.54	sequencetools.iterate_sequence_pairwise_wrapped . . . . .	909
29.1.55	sequencetools.join_subsequences . . . . .	909
29.1.56	sequencetools.join_subsequences_by_sign_of_subsequence_elements . . . . .	909
29.1.57	sequencetools.map_sequence_elements_to_canonic_tuples . . . . .	909
29.1.58	sequencetools.map_sequence_elements_to_numbered_sublists . . . . .	910
29.1.59	sequencetools.merge_duration_sequences . . . . .	910
29.1.60	sequencetools.negate_absolute_value_of_sequence_elements_at_indices . . . . .	910
29.1.61	sequencetools.negate_absolute_value_of_sequence_elements_cyclically . . . . .	910
29.1.62	sequencetools.negate_sequence_elements_at_indices . . . . .	911
29.1.63	sequencetools.negate_sequence_elements_cyclically . . . . .	911
29.1.64	sequencetools.overwrite_sequence_elements_at_indices . . . . .	911
29.1.65	sequencetools.pair_duration_sequence_elements_with_input_pair_values . . . . .	911
29.1.66	sequencetools.partition_sequence_by_backgrounded_weights . . . . .	912
29.1.67	sequencetools.partition_sequence_by_counts . . . . .	912
29.1.68	sequencetools.partition_sequence_by_ratio_of_lengths . . . . .	914
29.1.69	sequencetools.partition_sequence_by_ratio_of_weights . . . . .	914
29.1.70	sequencetools.partition_sequence_by_restricted_growth_function . . . . .	915
29.1.71	sequencetools.partition_sequence_by_sign_of_elements . . . . .	915
29.1.72	sequencetools.partition_sequence_by_value_of_elements . . . . .	916
29.1.73	sequencetools.partition_sequence_by_weights_at_least . . . . .	916
29.1.74	sequencetools.partition_sequence_by_weights_at_most . . . . .	916
29.1.75	sequencetools.partition_sequence_by_weights_exactly . . . . .	917
29.1.76	sequencetools.partition_sequence_extended_to_counts . . . . .	918
29.1.77	sequencetools.permute_sequence . . . . .	918
29.1.78	sequencetools.remove_sequence_elements_at_indices . . . . .	918
29.1.79	sequencetools.remove_sequence_elements_at_indices_cyclically . . . . .	919
29.1.80	sequencetools.remove_subsequence_of_weight_at_index . . . . .	919
29.1.81	sequencetools.repeat_runs_in_sequence_to_count . . . . .	919
29.1.82	sequencetools.repeat_sequence_elements_at_indices . . . . .	920
29.1.83	sequencetools.repeat_sequence_elements_at_indices_cyclically . . . . .	920
29.1.84	sequencetools.repeat_sequence_elements_n_times_each . . . . .	920
29.1.85	sequencetools.repeat_sequence_n_times . . . . .	921
29.1.86	sequencetools.repeat_sequence_to_length . . . . .	921
29.1.87	sequencetools.repeat_sequence_to_weight_at_least . . . . .	921
29.1.88	sequencetools.repeat_sequence_to_weight_at_most . . . . .	921
29.1.89	sequencetools.repeat_sequence_to_weight_exactly . . . . .	922
29.1.90	sequencetools.replace_sequence_elements_cyclically_with_new_material . . . . .	922
29.1.91	sequencetools.retain_sequence_elements_at_indices . . . . .	922
29.1.92	sequencetools.retain_sequence_elements_at_indices_cyclically . . . . .	922
29.1.93	sequencetools.reverse_sequence . . . . .	923
29.1.94	sequencetools.reverse_sequence_elements . . . . .	923
29.1.95	sequencetools.rotate_sequence . . . . .	923
29.1.96	sequencetools.splice_new_elements_between_sequence_elements . . . . .	923
29.1.97	sequencetools.split_sequence_by_weights . . . . .	924

29.1.98	sequencetools.split_sequence_extended_to_weights	925
29.1.99	sequencetools.sum_consecutive_sequence_elements_by_sign	925
29.1.100	sequencetools.sum_sequence_elements_at_indices	925
29.1.101	sequencetools.truncate_runs_in_sequence	926
29.1.102	sequencetools.truncate_sequence_to_sum	926
29.1.103	sequencetools.truncate_sequence_to_weight	927
29.1.104	sequencetools.yield_all_combinations_of_sequence_elements	927
29.1.105	sequencetools.yield_all_k_ary_sequences_of_length	928
29.1.106	sequencetools.yield_all_pairs_between_sequences	928
29.1.107	sequencetools.yield_all_partitions_of_sequence	928
29.1.108	sequencetools.yield_all_permutations_of_sequence	928
29.1.109	sequencetools.yield_all_permutations_of_sequence_in_orbit	929
29.1.110	sequencetools.yield_all_restricted_growth_functions_of_length	929
29.1.111	sequencetools.yield_all_rotations_of_sequence	929
29.1.112	sequencetools.yield_all_set_partitions_of_sequence	929
29.1.113	sequencetools.yield_all_subsequences_of_sequence	930
29.1.114	sequencetools.yield_all_unordered_pairs_of_sequence	930
29.1.115	sequencetools.yield_outer_product_of_sequences	931
29.1.116	sequencetools.zip_sequences_cyclically	931
29.1.117	sequencetools.zip_sequences_without_truncation	932
<b>30</b>	<b>sievetools</b>	<b>933</b>
30.1	Concrete classes	933
30.1.1	sievetools.BaseResidueClass	933
30.1.2	sievetools.ResidueClass	934
30.1.3	sievetools.Sieve	936
<b>31</b>	<b>skiptools</b>	<b>939</b>
31.1	Concrete classes	939
31.1.1	skiptools.Skip	939
31.2	Functions	941
31.2.1	skiptools.make_repeated_skips_from_time_signatures	941
31.2.2	skiptools.make_skips_with_multiplied_durations	941
<b>32</b>	<b>spannertools</b>	<b>943</b>
32.1	Abstract classes	943
32.1.1	spannertools.DirectedSpanner	943
32.1.2	spannertools.Spanner	949
32.2	Concrete classes	955
32.2.1	spannertools.BeamSpanner	955
32.2.2	spannertools.BraceSpanner	961
32.2.3	spannertools.ComplexBeamSpanner	967
32.2.4	spannertools.ComplexGlissandoSpanner	974
32.2.5	spannertools.CrescendoSpanner	980
32.2.6	spannertools.DecrescendoSpanner	988
32.2.7	spannertools.DuratedComplexBeamSpanner	996
32.2.8	spannertools.DynamicTextSpanner	1004
32.2.9	spannertools.GlissandoSpanner	1010
32.2.10	spannertools.HairpinSpanner	1016
32.2.11	spannertools.HiddenStaffSpanner	1024
32.2.12	spannertools.HorizontalBracketSpanner	1030
32.2.13	spannertools.MeasuredComplexBeamSpanner	1036
32.2.14	spannertools.MultipartBeamSpanner	1044
32.2.15	spannertools.OctavationSpanner	1050
32.2.16	spannertools.PhrasingSlurSpanner	1057
32.2.17	spannertools.PianoPedalSpanner	1063
32.2.18	spannertools.SlurSpanner	1069
32.2.19	spannertools.StaffLinesSpanner	1075
32.2.20	spannertools.TextScriptSpanner	1081

32.2.21	spannertools.TextSpanner . . . . .	1087
32.2.22	spannertools.TieSpanner . . . . .	1093
32.2.23	spannertools.TrillSpanner . . . . .	1099
32.3	Functions . . . . .	1105
32.3.1	spannertools.make_dynamic_spanner_below_with_nib_at_right . . . . .	1105
32.3.2	spannertools.make_solid_text_spanner_with_nib . . . . .	1105
<b>33</b>	<b>stafftools</b>	<b>1107</b>
33.1	Concrete classes . . . . .	1107
33.1.1	stafftools.RhythmicStaff . . . . .	1107
33.1.2	stafftools.Staff . . . . .	1114
33.2	Functions . . . . .	1120
33.2.1	stafftools.make_rhythmic_sketch_staff . . . . .	1120
<b>34</b>	<b>stringtools</b>	<b>1121</b>
34.1	Functions . . . . .	1121
34.1.1	stringtools.add_terminal_newlines . . . . .	1121
34.1.2	stringtools.arg_to_bidirectional_direction_string . . . . .	1121
34.1.3	stringtools.arg_to_bidirectional_lilypond_symbol . . . . .	1121
34.1.4	stringtools.arg_to_tridirectional_direction_string . . . . .	1122
34.1.5	stringtools.arg_to_tridirectional_lilypond_symbol . . . . .	1122
34.1.6	stringtools.arg_to_tridirectional_ordinal_constant . . . . .	1123
34.1.7	stringtools.capitalize_string_start . . . . .	1123
34.1.8	stringtools.format_input_lines_as_doc_string . . . . .	1123
34.1.9	stringtools.format_input_lines_as_regression_test . . . . .	1124
34.1.10	stringtools.is_dash_case_file_name . . . . .	1124
34.1.11	stringtools.is_dash_case_string . . . . .	1125
34.1.12	stringtools.is_lower_camel_case_string . . . . .	1125
34.1.13	stringtools.is_snake_case_file_name . . . . .	1125
34.1.14	stringtools.is_snake_case_file_name_with_extension . . . . .	1125
34.1.15	stringtools.is_snake_case_package_name . . . . .	1126
34.1.16	stringtools.is_snake_case_string . . . . .	1126
34.1.17	stringtools.is_space_delimited_lowercase_string . . . . .	1126
34.1.18	stringtools.is_upper_camel_case_string . . . . .	1126
34.1.19	stringtools.pluralize_string . . . . .	1127
34.1.20	stringtools.snake_case_to_lower_camel_case . . . . .	1127
34.1.21	stringtools.snake_case_to_upper_camel_case . . . . .	1127
34.1.22	stringtools.space_delimited_lowercase_to_upper_camel_case . . . . .	1127
34.1.23	stringtools.string_to_accent_free_snake_case . . . . .	1127
34.1.24	stringtools.string_to_space_delimited_lowercase . . . . .	1128
34.1.25	stringtools.strip_diacritics_from_binary_string . . . . .	1128
34.1.26	stringtools.upper_camel_case_to_snake_case . . . . .	1128
34.1.27	stringtools.upper_camel_case_to_space_delimited_lowercase . . . . .	1129
<b>35</b>	<b>tempotools</b>	<b>1131</b>
35.1	Functions . . . . .	1131
35.1.1	tempotools.report_integer_tempo_rewrite_pairs . . . . .	1131
35.1.2	tempotools.rewrite_duration_under_new_tempo . . . . .	1131
35.1.3	tempotools.rewrite_integer_tempo . . . . .	1132
<b>36</b>	<b>timeintervaltools</b>	<b>1133</b>
36.1	Abstract classes . . . . .	1133
36.1.1	timeintervaltools.TimeIntervalAggregateMixin . . . . .	1133
36.1.2	timeintervaltools.TimeIntervalMixin . . . . .	1139
36.2	Concrete classes . . . . .	1141
36.2.1	timeintervaltools.TimeInterval . . . . .	1141
36.2.2	timeintervaltools.TimeIntervalTree . . . . .	1144
36.2.3	timeintervaltools.TimeIntervalTreeDictionary . . . . .	1160
36.2.4	timeintervaltools.TimeIntervalTreeNode . . . . .	1182

36.3	Functions	1183
36.3.1	timeintervaltools.concatenate_trees	1183
36.3.2	timeintervaltools.make_test_intervals	1183
36.3.3	timeintervaltools.mask_intervals_with_intervals	1183
36.3.4	timeintervaltools.resolve_overlaps_between_nonoverlapping_trees	1184
<b>37</b>	<b>timerelementtools</b>	<b>1185</b>
37.1	Abstract classes	1185
37.1.1	timerelementtools.TimeRelation	1185
37.2	Concrete classes	1188
37.2.1	timerelementtools.CompoundInequality	1188
37.2.2	timerelementtools.OffsetTimespanTimeRelation	1192
37.2.3	timerelementtools.SimpleInequality	1195
37.2.4	timerelementtools.TimespanTimespanTimeRelation	1197
37.3	Functions	1202
37.3.1	timerelementtools.offset_happens_after_timespan_starts	1202
37.3.2	timerelementtools.offset_happens_after_timespan_stops	1202
37.3.3	timerelementtools.offset_happens_before_timespan_starts	1203
37.3.4	timerelementtools.offset_happens_before_timespan_stops	1204
37.3.5	timerelementtools.offset_happens_during_timespan	1204
37.3.6	timerelementtools.offset_happens_when_timespan_starts	1204
37.3.7	timerelementtools.offset_happens_when_timespan_stops	1205
37.3.8	timerelementtools.timespan_2_contains_timespan_1_improperly	1205
37.3.9	timerelementtools.timespan_2_curtails_timespan_1	1206
37.3.10	timerelementtools.timespan_2_delays_timespan_1	1206
37.3.11	timerelementtools.timespan_2_happens_during_timespan_1	1206
37.3.12	timerelementtools.timespan_2_intersects_timespan_1	1207
37.3.13	timerelementtools.timespan_2_is_congruent_to_timespan_1	1207
37.3.14	timerelementtools.timespan_2_overlaps_all_of_timespan_1	1208
37.3.15	timerelementtools.timespan_2_overlaps_only_start_of_timespan_1	1208
37.3.16	timerelementtools.timespan_2_overlaps_only_stop_of_timespan_1	1208
37.3.17	timerelementtools.timespan_2_overlaps_start_of_timespan_1	1209
37.3.18	timerelementtools.timespan_2_overlaps_stop_of_timespan_1	1209
37.3.19	timerelementtools.timespan_2_starts_after_timespan_1_starts	1209
37.3.20	timerelementtools.timespan_2_starts_after_timespan_1_stops	1210
37.3.21	timerelementtools.timespan_2_starts_before_timespan_1_starts	1210
37.3.22	timerelementtools.timespan_2_starts_before_timespan_1_stops	1210
37.3.23	timerelementtools.timespan_2_starts_during_timespan_1	1211
37.3.24	timerelementtools.timespan_2_starts_when_timespan_1_starts	1211
37.3.25	timerelementtools.timespan_2_starts_when_timespan_1_stops	1212
37.3.26	timerelementtools.timespan_2_stops_after_timespan_1_starts	1212
37.3.27	timerelementtools.timespan_2_stops_after_timespan_1_stops	1212
37.3.28	timerelementtools.timespan_2_stops_before_timespan_1_starts	1213
37.3.29	timerelementtools.timespan_2_stops_before_timespan_1_stops	1213
37.3.30	timerelementtools.timespan_2_stops_during_timespan_1	1213
37.3.31	timerelementtools.timespan_2_stops_when_timespan_1_starts	1214
37.3.32	timerelementtools.timespan_2_stops_when_timespan_1_stops	1214
37.3.33	timerelementtools.timespan_2_trisects_timespan_1	1214
<b>38</b>	<b>timesignaturetools</b>	<b>1215</b>
38.1	Concrete classes	1215
38.1.1	timesignaturetools.MetricalHierarchy	1215
38.1.2	timesignaturetools.MetricalHierarchyInventory	1221
38.1.3	timesignaturetools.MetricalKernel	1225
38.2	Functions	1227
38.2.1	timesignaturetools.duration_and_possible_denominators_to_time_signature	1227
38.2.2	timesignaturetools.establish_metrical_hierarchy	1227
38.2.3	timesignaturetools.fit_metrical_hierarchies_to_expr	1232

38.2.4	timesignaturetools.make_gridded_test_rhythm . . . . .	1233
<b>39</b>	<b>timespantools</b>	<b>1235</b>
39.1	Concrete classes . . . . .	1235
39.1.1	timespantools.Timespan . . . . .	1235
39.1.2	timespantools.TimespanInventory . . . . .	1255
<b>40</b>	<b>tonalanalysistools</b>	<b>1285</b>
40.1	Concrete classes . . . . .	1285
40.1.1	tonalanalysistools.ChordClass . . . . .	1285
40.1.2	tonalanalysistools.ChordQualityIndicator . . . . .	1289
40.1.3	tonalanalysistools.ExtentIndicator . . . . .	1292
40.1.4	tonalanalysistools.InversionIndicator . . . . .	1293
40.1.5	tonalanalysistools.Mode . . . . .	1294
40.1.6	tonalanalysistools.OmissionIndicator . . . . .	1295
40.1.7	tonalanalysistools.QualityIndicator . . . . .	1296
40.1.8	tonalanalysistools.RomanNumeral . . . . .	1297
40.1.9	tonalanalysistools.Scale . . . . .	1298
40.1.10	tonalanalysistools.ScaleDegree . . . . .	1302
40.1.11	tonalanalysistools.SuspensionIndicator . . . . .	1303
40.1.12	tonalanalysistools.TonalAnalysisAgent . . . . .	1304
40.2	Functions . . . . .	1307
40.2.1	tonalanalysistools.select . . . . .	1307
<b>41</b>	<b>tuplettools</b>	<b>1309</b>
41.1	Concrete classes . . . . .	1309
41.1.1	tuplettools.FixedDurationTuplet . . . . .	1309
41.1.2	tuplettools.Tuplet . . . . .	1326
<b>42</b>	<b>voicetools</b>	<b>1343</b>
42.1	Concrete classes . . . . .	1343
42.1.1	voicetools.Voice . . . . .	1343
<b>43</b>	<b>wellformednesstools</b>	<b>1351</b>
43.1	Abstract classes . . . . .	1351
43.1.1	wellformednesstools.Check . . . . .	1351
43.2	Concrete classes . . . . .	1352
43.2.1	wellformednesstools.BeamedQuarterNoteCheck . . . . .	1352
43.2.2	wellformednesstools.DiscontiguousSpannerCheck . . . . .	1353
43.2.3	wellformednesstools.DuplicateIdCheck . . . . .	1354
43.2.4	wellformednesstools.EmptyContainerCheck . . . . .	1355
43.2.5	wellformednesstools.IntermarkedHairpinCheck . . . . .	1356
43.2.6	wellformednesstools.MisduratedMeasureCheck . . . . .	1358
43.2.7	wellformednesstools.MisfilledMeasureCheck . . . . .	1359
43.2.8	wellformednesstools.MispitchedTieCheck . . . . .	1360
43.2.9	wellformednesstools.MisrepresentedFlagCheck . . . . .	1361
43.2.10	wellformednesstools.MissingParentCheck . . . . .	1362
43.2.11	wellformednesstools.NestedMeasureCheck . . . . .	1363
43.2.12	wellformednesstools.OverlappingBeamCheck . . . . .	1364
43.2.13	wellformednesstools.OverlappingGlissandoCheck . . . . .	1365
43.2.14	wellformednesstools.OverlappingOctavationCheck . . . . .	1366
43.2.15	wellformednesstools.ShortHairpinCheck . . . . .	1367
<b>II</b>	<b>Demos and example packages</b>	<b>1369</b>
<b>44</b>	<b>desordre</b>	<b>1371</b>
44.1	Functions . . . . .	1371
44.1.1	desordre.make_desordre_cell . . . . .	1371



44.1.2	desordre.make_desordre_lilypond_file . . . . .	1371
44.1.3	desordre.make_desordre_measure . . . . .	1371
44.1.4	desordre.make_desordre_pitches . . . . .	1371
44.1.5	desordre.make_desordre_score . . . . .	1371
44.1.6	desordre.make_desordre_staff . . . . .	1371
<b>45</b>	<b>ferneyhough</b>	<b>1373</b>
45.1	Functions . . . . .	1373
45.1.1	ferneyhough.configure_lilypond_file . . . . .	1373
45.1.2	ferneyhough.configure_score . . . . .	1373
45.1.3	ferneyhough.make_lilypond_file . . . . .	1373
45.1.4	ferneyhough.make_nested_tuplet . . . . .	1373
45.1.5	ferneyhough.make_row_of_nested_tuplets . . . . .	1373
45.1.6	ferneyhough.make_rows_of_nested_tuplets . . . . .	1373
45.1.7	ferneyhough.make_score . . . . .	1373
<b>46</b>	<b>mozart</b>	<b>1375</b>
46.1	Functions . . . . .	1375
46.1.1	mozart.choose_mozart_measures . . . . .	1375
46.1.2	mozart.make_mozart_lilypond_file . . . . .	1375
46.1.3	mozart.make_mozart_measure . . . . .	1375
46.1.4	mozart.make_mozart_measure_corpus . . . . .	1375
46.1.5	mozart.make_mozart_score . . . . .	1375
<b>47</b>	<b>part</b>	<b>1377</b>
47.1	Concrete classes . . . . .	1377
47.1.1	part.PartCantusScoreTemplate . . . . .	1377
47.2	Functions . . . . .	1378
47.2.1	part.add_bell_music_to_score . . . . .	1378
47.2.2	part.add_string_music_to_score . . . . .	1378
47.2.3	part.apply_bowing_marks . . . . .	1378
47.2.4	part.apply_dynamic_marks . . . . .	1378
47.2.5	part.apply_expressive_marks . . . . .	1378
47.2.6	part.apply_final_bar_lines . . . . .	1378
47.2.7	part.apply_page_breaks . . . . .	1378
47.2.8	part.apply_rehearsal_marks . . . . .	1378
47.2.9	part.configure_lilypond_file . . . . .	1378
47.2.10	part.configure_score . . . . .	1378
47.2.11	part.create_pitch_contour_reservoir . . . . .	1379
47.2.12	part.durate_pitch_contour_reservoir . . . . .	1379
47.2.13	part.edit_bass_voice . . . . .	1379
47.2.14	part.edit_cello_voice . . . . .	1379
47.2.15	part.edit_first_violin_voice . . . . .	1379
47.2.16	part.edit_second_violin_voice . . . . .	1379
47.2.17	part.edit_viola_voice . . . . .	1379
47.2.18	part.make_part_lilypond_file . . . . .	1379
47.2.19	part.shadow_pitch_contour_reservoir . . . . .	1379
<b>III</b>	<b>Abjad internal packages</b>	<b>1381</b>
<b>48</b>	<b>abctools</b>	<b>1383</b>
48.1	Abstract classes . . . . .	1383
48.1.1	abctools.ContextManager . . . . .	1383
48.1.2	abctools.Maker . . . . .	1384
48.1.3	abctools.Parser . . . . .	1385
48.2	Concrete classes . . . . .	1387
48.2.1	abctools.AbjadObject . . . . .	1387

<b>49</b>	<b>abjadbooktools</b>	<b>1389</b>
49.1	Abstract classes	1389
49.1.1	abjadbooktools.OutputFormat	1389
49.2	Concrete classes	1390
49.2.1	abjadbooktools.AbjadBookProcessor	1390
49.2.2	abjadbooktools.AbjadBookScript	1392
49.2.3	abjadbooktools.CodeBlock	1393
49.2.4	abjadbooktools.HTMLOutputFormat	1394
49.2.5	abjadbooktools.LaTeXOutputFormat	1396
49.2.6	abjadbooktools.ReSTOutputFormat	1397
<b>50</b>	<b>configurationtools</b>	<b>1399</b>
50.1	Abstract classes	1399
50.1.1	configurationtools.Configuration	1399
50.2	Concrete classes	1401
50.2.1	configurationtools.AbjadConfiguration	1401
<b>51</b>	<b>datastructuretools</b>	<b>1407</b>
51.1	Abstract classes	1407
51.1.1	datastructuretools.TypedCollection	1407
51.2	Concrete classes	1409
51.2.1	datastructuretools.BreakPointFunction	1409
51.2.2	datastructuretools.CyclicList	1416
51.2.3	datastructuretools.CyclicMatrix	1419
51.2.4	datastructuretools.CyclicPayloadTree	1421
51.2.5	datastructuretools.CyclicTuple	1434
51.2.6	datastructuretools.Matrix	1436
51.2.7	datastructuretools.OrdinalConstant	1438
51.2.8	datastructuretools.PayloadTree	1439
51.2.9	datastructuretools.SortedCollection	1452
51.2.10	datastructuretools.TreeContainer	1454
51.2.11	datastructuretools.TreeNode	1463
51.2.12	datastructuretools.TypedCounter	1466
51.2.13	datastructuretools.TypedFrozenSet	1468
51.2.14	datastructuretools.TypedList	1470
51.2.15	datastructuretools.TypedTuple	1475
<b>52</b>	<b>decoratortools</b>	<b>1477</b>
52.1	Functions	1477
52.1.1	decoratortools.requires	1477
<b>53</b>	<b>developerscripttools</b>	<b>1479</b>
53.1	Abstract classes	1479
53.1.1	developerscripttools.DeveloperScript	1479
53.1.2	developerscripttools.DirectoryScript	1481
53.2	Concrete classes	1483
53.2.1	developerscripttools.AbjDevScript	1483
53.2.2	developerscripttools.AbjGrepScript	1485
53.2.3	developerscripttools.AbjUpScript	1487
53.2.4	developerscripttools.BuildApiScript	1489
53.2.5	developerscripttools.CleanScript	1491
53.2.6	developerscripttools.CountLinewidthsScript	1493
53.2.7	developerscripttools.CountToolsScript	1495
53.2.8	developerscripttools.MakeNewClassTemplateScript	1497
53.2.9	developerscripttools.MakeNewFunctionTemplateScript	1499
53.2.10	developerscripttools.PyTestScript	1501
53.2.11	developerscripttools.RenameModulesScript	1503
53.2.12	developerscripttools.ReplaceInFilesScript	1505
53.2.13	developerscripttools.RunDoctestsScript	1507



53.2.14	developerscripttools.SvnAddAllScript	1509
53.2.15	developerscripttools.SvnCommitScript	1511
53.2.16	developerscripttools.SvnMessageScript	1513
53.2.17	developerscripttools.SvnUpdateScript	1515
53.2.18	developerscripttools.TestAndRebuildScript	1517
53.3	Functions	1518
53.3.1	developerscripttools.get_developer_script_classes	1518
53.3.2	developerscripttools.run_abjadbook	1518
53.3.3	developerscripttools.run_abjdev	1518
<b>54</b>	<b>documentationtools</b>	<b>1519</b>
54.1	Abstract classes	1519
54.1.1	documentationtools.Documenter	1519
54.1.2	documentationtools.GraphvizObject	1521
54.1.3	documentationtools.ReSTDirective	1522
54.2	Concrete classes	1531
54.2.1	documentationtools.AbjadAPIGenerator	1531
54.2.2	documentationtools.ClassCrawler	1532
54.2.3	documentationtools.ClassDocumenter	1533
54.2.4	documentationtools.FunctionCrawler	1536
54.2.5	documentationtools.FunctionDocumenter	1538
54.2.6	documentationtools.GraphvizEdge	1540
54.2.7	documentationtools.GraphvizGraph	1541
54.2.8	documentationtools.GraphvizNode	1552
54.2.9	documentationtools.GraphvizSubgraph	1555
54.2.10	documentationtools.InheritanceGraph	1564
54.2.11	documentationtools.ModuleCrawler	1566
54.2.12	documentationtools.Pipe	1567
54.2.13	documentationtools.ReSTAutodocDirective	1569
54.2.14	documentationtools.ReSTAutosummaryDirective	1578
54.2.15	documentationtools.ReSTAutosummaryItem	1587
54.2.16	documentationtools.ReSTDDocument	1590
54.2.17	documentationtools.ReSTHeading	1599
54.2.18	documentationtools.ReSTHorizontalRule	1603
54.2.19	documentationtools.ReSTInheritanceDiagram	1606
54.2.20	documentationtools.ReSTLineageDirective	1615
54.2.21	documentationtools.ReSTOnlyDirective	1624
54.2.22	documentationtools.ReSTParagraph	1633
54.2.23	documentationtools.ReSTTOCDirective	1637
54.2.24	documentationtools.ReSTTOCItem	1646
54.2.25	documentationtools.ToolsPackageDocumenter	1649
54.3	Functions	1651
54.3.1	documentationtools.compare_images	1651
54.3.2	documentationtools.make_ligeti_example_lilypond_file	1651
54.3.3	documentationtools.make_reference_manual_graphviz_graph	1651
54.3.4	documentationtools.make_reference_manual_lilypond_file	1651
54.3.5	documentationtools.make_text_alignment_example_lilypond_file	1651
<b>55</b>	<b>exceptiontools</b>	<b>1653</b>
55.1	Concrete classes	1653
55.1.1	exceptiontools.AssignabilityError	1653
55.1.2	exceptiontools.ExtraMarkError	1654
55.1.3	exceptiontools.ExtraNamedComponentError	1655
55.1.4	exceptiontools.ExtraNoteHeadError	1656
55.1.5	exceptiontools.ExtraPitchError	1657
55.1.6	exceptiontools.ExtraSpannerError	1658
55.1.7	exceptiontools.ImpreciseTempoError	1659
55.1.8	exceptiontools.InstrumentError	1660

55.1.9	exceptiontools.LilyPondParserError	1661
55.1.10	exceptiontools.MissingInstrumentError	1662
55.1.11	exceptiontools.MissingMarkError	1663
55.1.12	exceptiontools.MissingMeasureError	1664
55.1.13	exceptiontools.MissingNamedComponentError	1665
55.1.14	exceptiontools.MissingNoteHeadError	1666
55.1.15	exceptiontools.MissingPitchError	1667
55.1.16	exceptiontools.MissingSpannerError	1668
55.1.17	exceptiontools.MissingTempoError	1669
55.1.18	exceptiontools.OverfullContainerError	1670
55.1.19	exceptiontools.PartitionError	1671
55.1.20	exceptiontools.SchemeParserFinishedException	1672
55.1.21	exceptiontools.SpannerPopulationError	1673
55.1.22	exceptiontools.TempoError	1674
55.1.23	exceptiontools.TieChainError	1675
55.1.24	exceptiontools.TimeSignatureError	1676
55.1.25	exceptiontools.TonalHarmonyError	1677
55.1.26	exceptiontools.TupletFuseError	1678
55.1.27	exceptiontools.UnboundedTimeIntervalError	1679
55.1.28	exceptiontools.UnderfullContainerError	1680
<b>56</b>	<b>formattools</b>	<b>1683</b>
56.1	Functions	1683
56.1.1	formattools.format_lilypond_attribute	1683
56.1.2	formattools.format_lilypond_value	1683
56.1.3	formattools.get_all_format_contributions	1683
56.1.4	formattools.get_all_mark_format_contributions	1683
56.1.5	formattools.get_articulation_format_contributions	1683
56.1.6	formattools.get_comment_format_contributions_for_slot	1684
56.1.7	formattools.get_context_mark_format_contributions_for_slot	1684
56.1.8	formattools.get_context_mark_format_pieces	1684
56.1.9	formattools.get_context_setting_format_contributions	1684
56.1.10	formattools.get_grob_override_format_contributions	1684
56.1.11	formattools.get_grob_revert_format_contributions	1684
56.1.12	formattools.get_lilypond_command_mark_format_contributions_for_slot	1684
56.1.13	formattools.get_markup_format_contributions	1684
56.1.14	formattools.get_spanner_format_contributions	1685
56.1.15	formattools.get_stem_tremolo_format_contributions	1685
56.1.16	formattools.is_formattable_context_mark_for_component	1685
56.1.17	formattools.make_lilypond_override_string	1685
56.1.18	formattools.make_lilypond_revert_string	1685
56.1.19	formattools.report_component_format_contributions	1685
56.1.20	formattools.report_spanner_format_contributions	1686
<b>57</b>	<b>importtools</b>	<b>1687</b>
57.1	Concrete classes	1687
57.1.1	importtools.ImportManager	1687
<b>58</b>	<b>introspectiontools</b>	<b>1689</b>
58.1	Functions	1689
58.1.1	introspectiontools.class_to_tools_package_qualified_class_name	1689
58.1.2	introspectiontools.get_current_function_name	1689
<b>59</b>	<b>lilypondparsertools</b>	<b>1691</b>
59.1	Abstract classes	1691
59.1.1	lilypondparsertools.Music	1691
59.1.2	lilypondparsertools.SimultaneousMusic	1692
59.2	Concrete classes	1693
59.2.1	lilypondparsertools.ContextSpeccedMusic	1693

59.2.2	lilypondparsertools.GuileProxy . . . . .	1694
59.2.3	lilypondparsertools.LilyPondDuration . . . . .	1696
59.2.4	lilypondparsertools.LilyPondEvent . . . . .	1697
59.2.5	lilypondparsertools.LilyPondFraction . . . . .	1698
59.2.6	lilypondparsertools.LilyPondGrammarGenerator . . . . .	1699
59.2.7	lilypondparsertools.LilyPondLexicalDefinition . . . . .	1700
59.2.8	lilypondparsertools.LilyPondParser . . . . .	1703
59.2.9	lilypondparsertools.LilyPondSyntacticalDefinition . . . . .	1712
59.2.10	lilypondparsertools.ReducedLyParser . . . . .	1729
59.2.11	lilypondparsertools.SchemeParser . . . . .	1735
59.2.12	lilypondparsertools.SequentialMusic . . . . .	1738
59.2.13	lilypondparsertools.SyntaxNode . . . . .	1739
59.3	Functions . . . . .	1740
59.3.1	lilypondparsertools.parse_reduced_ly_syntax . . . . .	1740
<b>60</b>	<b>lilypondproxytools</b>	<b>1741</b>
60.1	Concrete classes . . . . .	1741
60.1.1	lilypondproxytools.LilyPondComponentPlugIn . . . . .	1741
60.1.2	lilypondproxytools.LilyPondContextProxy . . . . .	1742
60.1.3	lilypondproxytools.LilyPondContextSettingComponentPlugIn . . . . .	1743
60.1.4	lilypondproxytools.LilyPondGrobOverrideComponentPlugIn . . . . .	1744
60.1.5	lilypondproxytools.LilyPondGrobProxy . . . . .	1745
60.1.6	lilypondproxytools.LilyPondGrobProxyContextWrapper . . . . .	1746
60.1.7	lilypondproxytools.LilyPondObjectProxy . . . . .	1747
60.1.8	lilypondproxytools.LilyPondTweakReservoir . . . . .	1748
<b>61</b>	<b>testtools</b>	<b>1749</b>
61.1	Concrete classes . . . . .	1749
61.1.1	testtools.BenchmarkScoreMaker . . . . .	1749
61.2	Functions . . . . .	1753
61.2.1	testtools.apply_additional_layout . . . . .	1753
61.2.2	testtools.compare . . . . .	1753
61.2.3	testtools.read_test_output . . . . .	1753
61.2.4	testtools.write_test_output . . . . .	1753
<b>62</b>	<b>updatetools</b>	<b>1755</b>
62.1	Concrete classes . . . . .	1755
62.1.1	updatetools.UpdateManager . . . . .	1755
<b>Index</b>		<b>1757</b>



## **Part I**

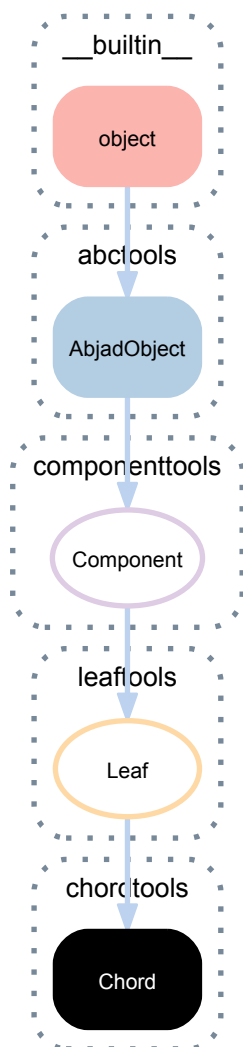
# **Core composition packages**



# CHORDTOOLS

## 1.1 Concrete classes

### 1.1.1 chordtools.Chord



**class** chordtools.**Chord** (\*args, \*\*kwargs)

A chord.

**Example.**

```
>>> chord = Chord("<e' cs'' f''>4")
>>> show(chord)
```



## Bases

- `leaftools.Leaf`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

**Chord.lilypond\_format**  
LilyPond format of chord.

**Example.**

```
>>> chord = Chord("<e' cs'' f''>4")
>>> show(chord)
```



```
>>> chord.lilypond_format
"<e' cs'' f''>4"
```

Returns string.

**Chord.override**  
LilyPond grob override component plug-in.

**Example.** Override LilyPond accidental, note head and stem grobs:

```
>>> chord = Chord("<e' cs'' f''>4")
>>> show(chord)
```



```
>>> chord.override.accidental.color = 'red'
>>> chord.override.note_head.color = 'red'
>>> chord.override.stem.color = 'red'
>>> show(chord)
```



Returns none.

**Chord.set**  
LilyPond context setting component plug-in.

**Example.** Set LilyPond `stemLeftBeamCount` and `stemRightBeamCount` context settings:

```
>>> chord = Chord("<e' cs'' f''>16")
>>> show(chord)
```





```
>>> beam = spannertools.BeamSpanner()
>>> beam.attach([chord])
>>> chord.set.stem_left_beam_count = 0
>>> chord.set.stem_right_beam_count = 2
>>> show(chord)
```



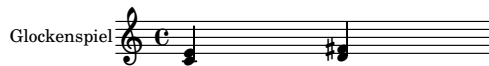
Returns none.

#### **Chord.sounding\_pitches**

Sounding pitches in chord.

##### **Example.**

```
>>> staff = Staff("<c' e'>4 <d' fs'>4")
>>> glockenspiel = instrumenttools.Glockenspiel()(staff)
>>> instrumenttools.transpose_from_sounding_pitch_to_written_pitch(
...     staff)
>>> show(staff)
```



```
>>> staff[0].sounding_pitches
(NamedPitch("c'"), NamedPitch("e'"))
```

Returns tuple.

#### **(Component).storage\_format**

Storage format of component.

Returns string.

## **Read/write properties**

#### **(Leaf).lilypond\_duration\_multiplier**

LilyPond duration multiplier.

Set to positive multiplier or none.

Returns positive multiplier or none.

#### **Chord.note\_heads**

Note heads in chord.

##### **Example 1.** Get note heads in chord:

```
>>> chord = Chord("<g' c' e'>4")
>>> show(chord)
```



```
>>> chord.note_heads
(NoteHead("g'"), NoteHead("c'"), NoteHead("e'"))
```

##### **Example 2.** Set note heads with pitch names:

```
>>> chord = Chord("<g' c' e'>4")
>>> show(chord)
```



```
>>> chord.note_heads = "c' d' fs'"
>>> show(chord)
```



**Example 3.** Set note heads with pitch numbers:

```
>>> chord = Chord("<g' c' ' e' '>4")
>>> show(chord)
```



```
>>> chord.note_heads = [16, 17, 19]
>>> show(chord)
```



Set note heads with any iterable.

Returns tuple.

`Chord.written_duration`

Written duration of chord.

**Example 1.** Get written duration:

```
>>> chord = Chord("<e' cs' ' f' '>4")
>>> show(chord)
```



```
>>> chord.written_duration
Duration(1, 4)
```

**Example 2.** Set written duration:

```
>>> chord = Chord("<e' cs' ' f' '>4")
>>> show(chord)
```



```
>>> chord.written_duration = Duration(1, 16)
>>> show(chord)
```



Set duration.

Returns duration.

`(Leaf).written_pitch_indication_is_at_sounding_pitch`

Returns true when written pitch is at sounding pitch. Returns false when written pitch is transposed.

`(Leaf).written_pitch_indication_is_nonsemantic`

Returns true when pitch is nonsemantic. Returns false otherwise.

Set to true when using leaves only graphically.

Setting this value to true sets sounding pitch indicator to false.

`Chord.written_pitches`

Written pitches in chord.

**Example 1.** Get written pitches:

```
>>> chord = Chord("<g' c' e'>4")
>>> show(chord)
```



```
>>> for written_pitch in chord.written_pitches:
...     written_pitch
NamedPitch("g' ")
NamedPitch("c' ")
NamedPitch("e' ")
```

**Example 2.** Set written pitches with pitch names:

```
>>> chord = Chord("<e' g' c'>4")
>>> show(chord)
```



```
>>> chord.written_pitches = "f' b' d'"
>>> show(chord)
```



Set written pitches with any iterable.

Returns tuple.

## Methods

`Chord.append(note_head)`

Appends *note\_head* to chord.

**Example.**

```
>>> chord = Chord("<e' cs' f'>4")
>>> show(chord)
```



```
>>> chord.append("g' ")
>>> show(chord)
```



Set *note\_head* to a pitch, pitch name, pitch number or note head.

Sorts note heads automatically.

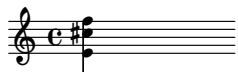
Returns none.

`Chord.extend(note_heads)`

Extends chord with *note\_heads*.

**Example 1.** Extend chord with pitch names:

```
>>> chord = Chord("<e' cs' f'>4")
>>> show(chord)
```



```
>>> chord.extend("d' c' fs'")
>>> show(chord)
```



**Example 2.** Extend chord with pitch numbers:

```
>>> chord = Chord("<e' cs' f'>4")
>>> show(chord)
```



```
>>> chord.extend([2, 12, 18])
>>> show(chord)
```



Sorts note heads automatically after execution.

Returns none.

`Chord.get_note_head(pitch)`  
Gets note head in chord by *pitch*.

**Example 1.** Get note head by pitch name:

```
>>> chord = Chord("<e' cs' f'>4")
>>> show(chord)
```



```
>>> note_head = chord.get_note_head("e'")
>>> note_head.tweak.color = 'red'
>>> show(chord)
```

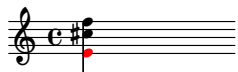


**Example 2.** Get note head by pitch number:

```
>>> chord = Chord("<e' cs' f'>4")
>>> show(chord)
```



```
>>> note_head = chord.get_note_head(4)
>>> note_head.tweak.color = 'red'
>>> show(chord)
```



Raises missing note head error when chord contains no note head with *pitch*.

Raises extra note head error when chord contains more than one note head with *pitch*.

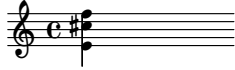
Returns note head.

`Chord.pop (i=-1)`

Pops note head *i* from chord.

**Example.**

```
>>> chord = Chord("<e' cs' ' f' '>4")
>>> show(chord)
```



```
>>> chord.pop(0)
NoteHead("e' ")
>>> show(chord)
```



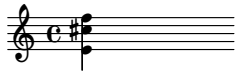
Returns note head.

`Chord.remove (note_head)`

Removes *note\_head* from chord.

**Example.**

```
>>> chord = Chord("<e' cs' ' f' '>4")
>>> show(chord)
```



```
>>> chord.remove(chord[0])
>>> show(chord)
```



Returns none.

`(Component).select (sequential=False)`

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

## Special methods

`Chord.__contains__ (expr)`

Returns true when *expr* equals one of the note heads in chord. Otherwise false.

`(Component).__copy__ (*args)`

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

`Chord.__delitem__ (i)`

Deletes note head *i* from chord.

Returns none.

(AbjadObject) .**\_\_eq\_\_**(*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

Chord.**\_\_getitem\_\_**(*i*)  
Gets note head *i* from chord.  
Returns note head.

Chord.**\_\_len\_\_**()  
Number of note heads in chord.  
Returns nonnegative integer.

(Component) .**\_\_mul\_\_**(*n*)  
Copies component *n* times and detaches spanners.  
Returns list of new components.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(Leaf) .**\_\_repr\_\_**()  
Interpreter representation of leaf.  
Returns string.

(Component) .**\_\_rmul\_\_**(*n*)  
Copies component *n* times and detach spanners.  
Returns list of new components.

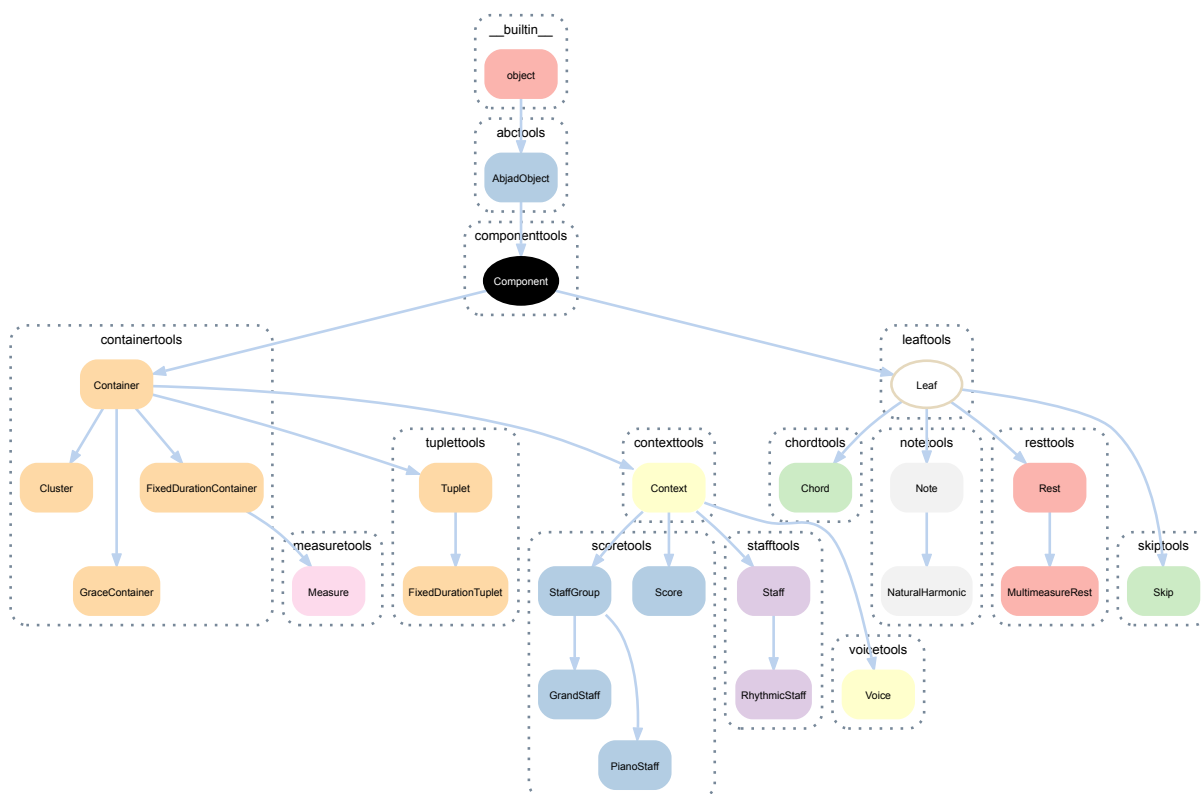
Chord.**\_\_setitem\_\_**(*i*, *expr*)  
Sets chord note head *i* to *expr*.  
Returns none.

(Leaf) .**\_\_str\_\_**()  
String representation of leaf.  
Returns string.

# COMPONENTTOOLS

## 2.1 Abstract classes

### 2.1.1 componenttools.Component



**class** `componenttools.Component`

Any score component.

Notes, rests, chords, tuplets, voices, staves and scores are all components.

### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`Component.lilypond_format`  
Lilypond format of component.

Returns string.

`Component.override`  
LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`Component.set`  
LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

`Component.storage_format`  
Storage format of component.

Returns string.

## Methods

`Component.select (sequential=False)`  
Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

## Special methods

`Component.__copy__ (*args)`  
Copies component with marks but without children of component or spanners attached to component.  
Returns new component.

`(AbjadObject).__eq__ (expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`Component.__mul__ (n)`  
Copies component *n* times and detaches spanners.  
Returns list of new components.

`(AbjadObject).__ne__ (expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__ ()`  
Interpreter representation of Abjad object.  
Returns string.

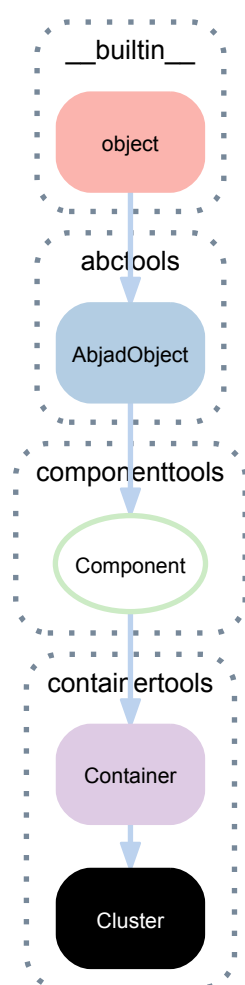
`Component.__rmul__ (n)`  
Copies component *n* times and detach spanners.  
Returns list of new components.



# CONTAINERTOOLS

## 3.1 Concrete classes

### 3.1.1 `containertools.Cluster`



**class** `containertools.Cluster` (*music=None*, *\*\*kwargs*)  
A cluster.

```
>>> cluster = containertools.Cluster("c'8 <d' g'>8 b'8")
```

```
>>> cluster
Cluster(c'8, <d' g'>8, b'8)
```

```
>>> show(cluster)
```



Returns cluster object.

## Bases

- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`Cluster.lilypond_format`

`(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

`(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

`(Container).is_simultaneous`

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

## Methods

(Container) .**append** (*component*)

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) .**extend** (*expr*)

Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) .**index** (*component*)

Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

(Container) **.pop** (*i=-1*)  
 Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

(Container) **.remove** (*component*)

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse** ()

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (*sequential=False*)

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)

Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords** ()

Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container).**\_\_contains\_\_**(*expr*)

True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component).**\_\_copy\_\_**(\*args)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(Container).**\_\_delitem\_\_**(*i*)

Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Returns none.

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container).**\_\_getitem\_\_**(*i*)

Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.

(Container).**\_\_len\_\_**()

Number of items in container.

Returns nonnegative integer.

(Component).**\_\_mul\_\_**(*n*)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

Cluster.**\_\_repr\_\_**()

(Component).**\_\_rmul\_\_**(*n*)

Copies component *n* times and detach spanners.

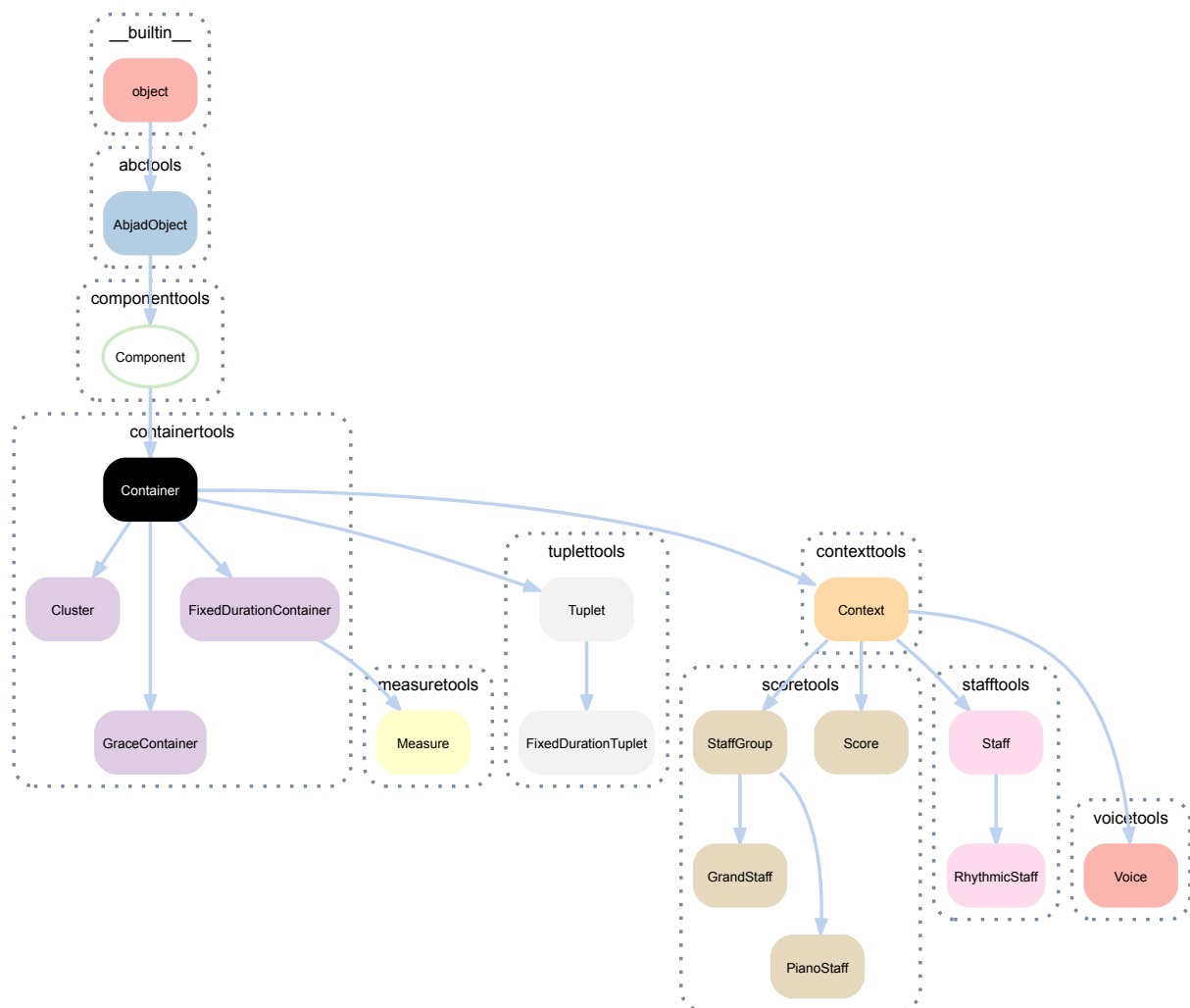
Returns list of new components.

(Container).**\_\_setitem\_\_**(*i*, *expr*)

Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

Returns none.

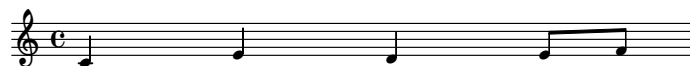
### 3.1.2 containertools.Container



**class** `containertools.Container` (*music=None*, *\*\*kwargs*)  
 An iterable container of music.

**Example:**

```
>>> container = Container("c'4 e'4 d'4 e'8 f'8")
>>> show(container)
```



#### Bases

- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(Component).lilypond_format`  
 Lilypond format of component.  
 Returns string.

(Component) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Component) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

(Component) **.storage\_format**

Storage format of component.

Returns string.

## Read/write properties

Container **.is\_simultaneous**

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

## Methods

Container **.append**(*component*)

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```





```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

`Container.extend(expr)`  
 Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



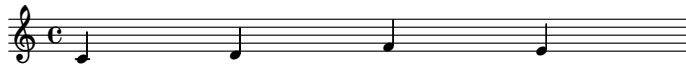
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

`Container.index(component)`  
 Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

`Container.insert(i, component, fracture_spanners=False)`  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```

**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

`Container.pop(i=-1)`

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

`Container.remove(component)`

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

`Container.reverse()`

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

`(Component).select(sequential=False)`

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

`Container.select_leaves(start=0, stop=None, leaf_classes=None, recurse=True, allow_discontiguous_leaves=False)`

Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

`Container.select_notes_and_chords()`

Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

`Container.__contains__(expr)`

True when *expr* appears in container. Otherwise false.

Returns boolean.

`(Component).__copy__(*args)`

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

`Container.__delitem__(i)`

Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Returns none.

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`Container.__getitem__(i)`

Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.

`Container.__len__()`

Number of items in container.

Returns nonnegative integer.

`(Component).__mul__(n)`

Copies component *n* times and detaches spanners.

Returns list of new components.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`Container.__repr__()`

Representation of container in Python interpreter.

Returns string.

`(Component).__rmul__(n)`

Copies component *n* times and detach spanners.

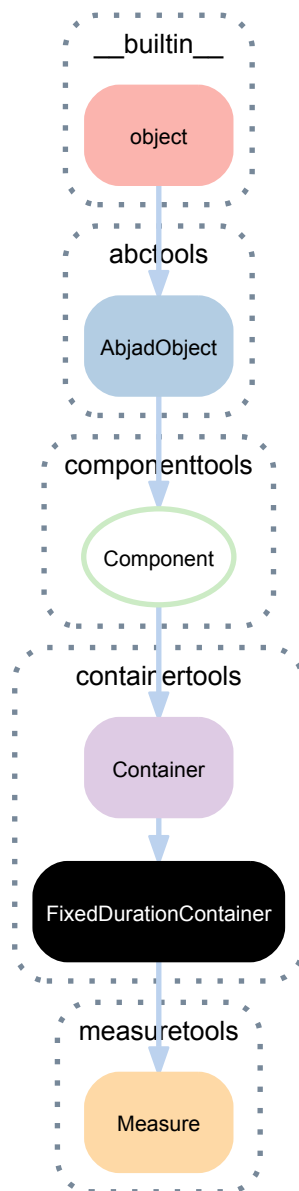
Returns list of new components.

`Container.__setitem__(i, expr)`

Set container *i* equal to *expr*. Find spanners that dominate `self[i]` and children of `self[i]`. Replace contents at `self[i]` with `'expr'`. Reattach spanners to new contents. This operation always leaves score tree in tact.

Returns none.

### 3.1.3 containertools.FixedDurationContainer



**class** `containertools.FixedDurationContainer` (*target\_duration*, *music=None*, *\*\*kwargs*)  
 A fixed-duration container.

```
>>> container = containertools.FixedDurationContainer(
...     (3, 8), "c'8 d'8 e'8")
```

```
>>> container
FixedDurationContainer(Duration(3, 8), [Note("c'8"), Note("d'8"), Note("e'8")])
```

Fixed-duration containers extend container behavior with format-time checking against a user-specified target duration.

Returns fixed-duration container.

#### Bases

- `containertools.Container`
- `componenttools.Component`

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`FixedDurationContainer.is_full`

True when preprolated duration equals target duration.

`FixedDurationContainer.is_misfilled`

True when preprolated duration does not equal target duration.

`FixedDurationContainer.is_overfull`

True when preprolated duration is greater than target duration.

`FixedDurationContainer.is_underfull`

True when preprolated duration is less than target duration.

`FixedDurationContainer.lilypond_format`

LilyPond format of fixed-duration container.

`(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

`(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

`(Container).is_simultaneous`

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

`FixedDurationContainer.target_duration`

Read / write target duration of fixed-duration container.

## Methods

`(Container).append(component)`

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

`(Container).extend(expr)`

Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```

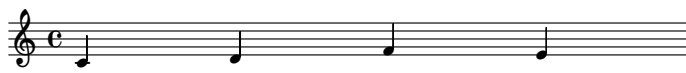


Returns none.

`(Container).index(component)`

Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

(Container) **.pop** (*i=-1*)  
 Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```





Returns component.

(Container) **.remove** (component)

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse** ()

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (sequential=False)

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False)

Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords** ()

Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container).**\_\_contains\_\_**(*expr*)

True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component).**\_\_copy\_\_**(\*args)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(Container).**\_\_delitem\_\_**(*i*)

Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Returns none.

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container).**\_\_getitem\_\_**(*i*)

Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.

(Container).**\_\_len\_\_**()

Number of items in container.

Returns nonnegative integer.

(Component).**\_\_mul\_\_**(*n*)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

FixedDurationContainer.**\_\_repr\_\_**()

(Component).**\_\_rmul\_\_**(*n*)

Copies component *n* times and detach spanners.

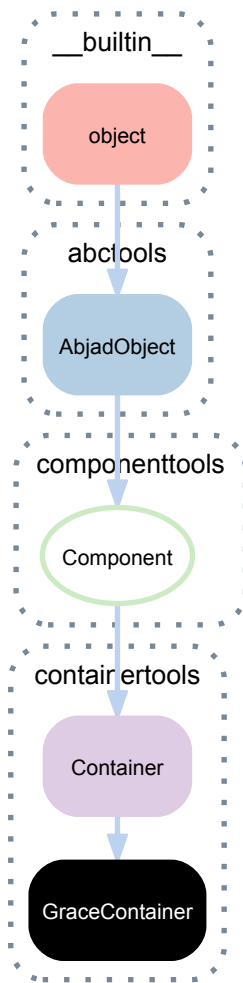
Returns list of new components.

(Container).**\_\_setitem\_\_**(*i*, *expr*)

Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

Returns none.

### 3.1.4 containertools.GraceContainer



**class** containertools.**GraceContainer** (*music=None, kind='grace', \*\*kwargs*)  
 A container of grace music.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> grace_notes = [Note("c'16"), Note("d'16")]
>>> containertools.GraceContainer(grace_notes, kind='grace')(voice[1])
Note("d'8")
>>> show(voice)
```



```
>>> after_grace_notes = [Note("e'16"), Note("f'16")]
>>> containertools.GraceContainer(
...     after_grace_notes, kind='after')(voice[1])
Note("d'8")
>>> show(voice)
```



Fill grace containers with notes, rests or chords.

Attach grace containers to nongrace notes, rests or chords.

## Bases

- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`GraceContainer.lilypond_format`

`(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

`(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

`(Container).is_simultaneous`

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

`GraceContainer.kind`

Gets *kind* of grace container.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> containertools.GraceContainer(
...     [Note("cs'16")], kind = 'grace')(staff[1])
Note("d'8")
>>> grace_container = staff[1].grace
>>> grace_container.kind
'grace'
```

Returns string.

Sets *kind* of grace container:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> containertools.GraceContainer(
...     [Note("cs'16")], kind = 'grace')(staff[1])
Note("d'8")
>>> grace_container = staff[1].grace
>>> grace_container.kind = 'acciaccatura'
>>> grace_container.kind
'acciaccatura'
```

Sets string.

Valid options include 'after', 'grace', 'acciaccatura', 'appoggiatura'.

## Methods

`(Container).append(component)`

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

`GraceContainer.attach(leaf)`

Attaches grace container to *leaf*.

Returns grace container.

`GraceContainer.detach()`

Detaches grace container from leaf.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> grace_container = containertools.GraceContainer(
...     [Note("cs'16")], kind = 'grace')
>>> grace_container(staff[1])
```

```
Note("d'8")
>>> f(staff)
\new Staff {
  c'8
  \grace {
    cs'16
  }
  d'8
  e'8
  f'8
}
```

```
>>> grace_container.detach()
GraceContainer()
>>> f(staff)
\new Staff {
  c'8
  d'8
  e'8
  f'8
}
```

Returns grace container.

(Container) **.extend** (*expr*)  
Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



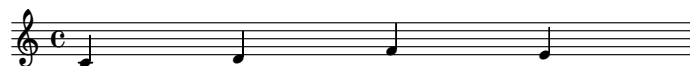
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) **.index** (*component*)  
Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
```

```
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2. Insert note. Fracture spanners:**

```
>>> container = Container([])
>>> container.extend("fsl6 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```

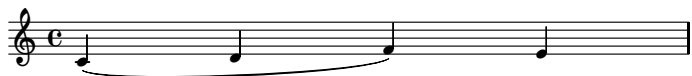


Returns none.

`(Container) .pop(i=-1)`

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

`(Container) .remove(component)`

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f' 4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse()**  
Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (*sequential=False*)  
Selects component.  
  
Returns component selection when *sequential* is false.  
  
Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)  
Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords()**  
Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

GraceContainer **.\_\_call\_\_** (*arg*)

(Container) **.\_\_contains\_\_** (*expr*)  
True when *expr* appears in container. Otherwise false.  
  
Returns boolean.



(Component) .**\_\_copy\_\_** (\*args)  
Copies component with marks but without children of component or spanners attached to component.  
Returns new component.

(Container) .**\_\_delitem\_\_** (i)  
Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners.  
Preserve spanners that component(s) cover(s).  
Returns none.

(AbjadObject) .**\_\_eq\_\_** (expr)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(Container) .**\_\_getitem\_\_** (i)  
Get container *i*. Shallow traversal of container for numeric indices only.  
Returns component.

(Container) .**\_\_len\_\_** ()  
Number of items in container.  
Returns nonnegative integer.

(Component) .**\_\_mul\_\_** (n)  
Copies component *n* times and detaches spanners.  
Returns list of new components.

(AbjadObject) .**\_\_ne\_\_** (expr)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

GraceContainer .**\_\_repr\_\_** ()

(Component) .**\_\_rmul\_\_** (n)  
Copies component *n* times and detach spanners.  
Returns list of new components.

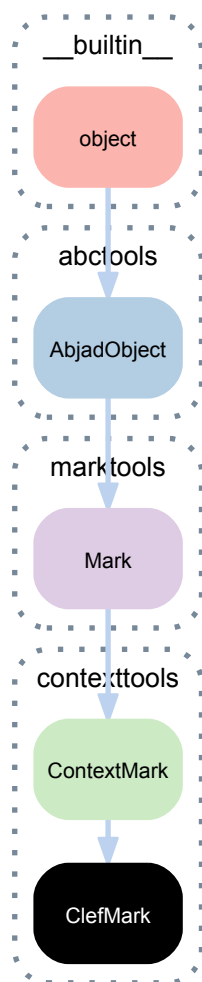
(Container) .**\_\_setitem\_\_** (i, expr)  
Set container *i* equal to *expr*. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.  
Returns none.



## CONTEXTTOOLS

### 4.1 Concrete classes

#### 4.1.1 contexttools.ClefMark



**class** contexttools.**ClefMark** (*clef\_name*, *target\_context=None*)  
A clef.

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")  
>>> show(staff)
```



```
>>> clef = contexttools.ClefMark('treble')(staff)
>>> clef = contexttools.ClefMark('alto')(staff[1])
>>> clef = contexttools.ClefMark('bass')(staff[2])
>>> clef = contexttools.ClefMark('treble^8')(staff[3])
>>> clef = contexttools.ClefMark('bass_8')(staff[4])
>>> clef = contexttools.ClefMark('tenor')(staff[5])
>>> clef = contexttools.ClefMark('bass^15')(staff[6])
>>> clef = contexttools.ClefMark('percussion')(staff[7])
>>> show(staff)
```



Clef marks target the staff context by default.

## Bases

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**

Effective context of context mark.

Returns context mark or none.

ClefMark **.lilypond\_format**

LilyPond format of clef.

```
>>> clef = contexttools.ClefMark('treble')
>>> clef.lilypond_format
'\clef "treble"'
```

Returns string.

ClefMark **.middle\_c\_position**

Middle C position of clef.

```
>>> clef = contexttools.ClefMark('treble')
>>> clef.middle_c_position
-6
```

Returns integer number of stafflines.

(Mark) **.start\_component**

Start component of mark.

Returns component or none.

ClefMark **.storage\_format**

Storage format of clef.

```
>>> print clef.storage_format
contexttools.ClefMark(
    'treble',
    target_context=stafftools.Staff
)
```

Returns string.

(ContextMark) **.target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

ClefMark **.clef\_name**

Gets and sets clef name.

```
>>> clef = contexttools.ClefMark('treble')
>>> clef.clef_name
'treble'
```

```
>>> clef.clef_name = 'alto'
>>> clef.clef_name
'alto'
```

Returns string.

## Methods

(ContextMark) **.attach**(start\_component)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Class methods

ClefMark **.list\_clef\_names**()

Lists clef names.

```
>>> for name in contexttools.ClefMark.list_clef_names():
...     name
...
'alto'
'baritone'
'bass'
'french'
'mezzosoprano'
'percussion'
'soprano'
'tab'
'tenor'
'treble'
'varbaritone'
```

Returns list of strings.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

`ClefMark.__copy__(*args)`  
Copies clef.

```
>>> import copy
>>> clef_1 = contexttools.ClefMark('alto')
>>> clef_2 = copy.copy(clef_1)
```

```
>>> clef_1, clef_2
(ClefMark('alto'), ClefMark('alto'))
```

```
>>> clef_1 == clef_2
True
```

```
>>> clef_1 is clef_2
False
```

Returns new clef.

`ClefMark.__eq__(arg)`  
True when clef name of *arg* equal clef name of clef. Otherwise false.

```
>>> clef_1 = contexttools.ClefMark('treble')
>>> clef_2 = contexttools.ClefMark('alto')
```

```
>>> clef_1 == clef_1
True
>>> clef_1 == clef_2
False
>>> clef_2 == clef_1
False
>>> clef_2 == clef_2
True
```

Returns boolean.

`ClefMark.__ne__(arg)`  
True when clef of *arg* does not equal clef name of clef. False otherwise.

```
>>> clef_1 = contexttools.ClefMark('treble')
>>> clef_2 = contexttools.ClefMark('alto')
```

```
>>> clef_1 != clef_1
False
>>> clef_1 != clef_2
True
>>> clef_2 != clef_1
True
>>> clef_2 != clef_2
False
```

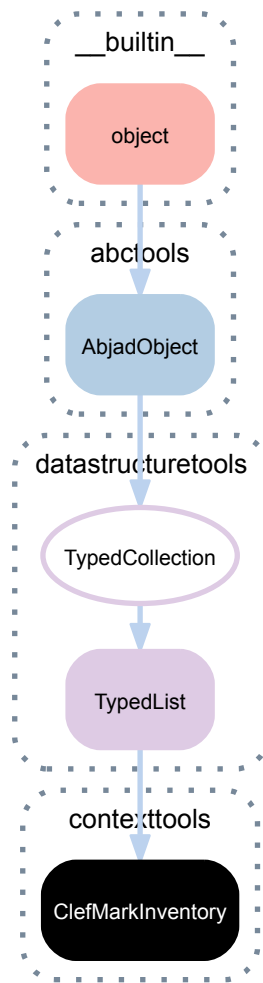
Returns boolean.

`ClefMark.__repr__()`  
Interpreter representation of clef.

```
>>> clef = contexttools.ClefMark('treble')
>>> clef
ClefMark('treble')
```

Returns string.

### 4.1.2 contexttools.ClefMarkInventory



**class** contexttools.**ClefMarkInventory** (*tokens=None, item\_class=None, name=None*)  
 An ordered list of clefs.

```
>>> inventory = contexttools.ClefMarkInventory(['treble', 'bass'])
```

```
>>> inventory
ClefMarkInventory([ClefMark('treble'), ClefMark('bass')])
```

```
>>> 'treble' in inventory
True
```

```
>>> contexttools.ClefMark('treble') in inventory
True
```

```
>>> 'alto' in inventory
False
```

Clef mark inventories implement list interface and are mutable.

#### Bases

- datastructuretools.TypedList
- datastructuretools.TypedCollection
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedList).append(token)`  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(  
...     item_class=int)  
>>> integer_collection[:]  
[]
```

```
>>> integer_collection.append('1')  
>>> integer_collection.append(2)  
>>> integer_collection.append(3.4)  
>>> integer_collection[:]  
[1, 2, 3]
```

Returns none.

`(TypedList).count(token)`  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(  
...     tokens=[0, 0., '0', 99],  
...     item_class=int)  
>>> integer_collection[:]  
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)  
3
```

Returns count.

`(TypedList).extend(tokens)`  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(  
...     item_class=int)  
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))  
>>> integer_collection[:]  
[0, 1, 2, 3]
```

Returns none.

`(TypedList).index(token)`  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(  
...     tokens=('c'qf', "as'", 'b', 'dss'),  
...     item_class=pitchtools.NamedPitch)  
>>> pitch_collection.index("as'")  
1
```

Returns index.



(TypedList) **.insert** (*i*, *token*)  
Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('1', 2, 4.3))
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None*, *item\_class=None*, *name=None*)

(TypedList) **.pop** (*i=-1*)  
Aliases list.pop().

(TypedList) **.remove** (*token*)  
Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()  
Aliases list.reverse().

(TypedList) **.sort** (*cmp=None*, *key=None*, *reverse=False*)  
Aliases list.sort().

## Special methods

(TypedCollection) **.\_\_contains\_\_** (*token*)

(TypedList) **.\_\_delitem\_\_** (*i*)  
Aliases list.\_\_delitem\_\_().

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedList) **.\_\_getitem\_\_** (*i*)  
Aliases list.\_\_getitem\_\_().

(TypedList) **.\_\_iadd\_\_** (*expr*)  
Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
```

```

        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'fff',
        target_context=stafftools.Staff
    )
],
item_class=contexttools.DynamicMark
)

```

Returns collection.

(TypedCollection) .**\_\_iter\_\_**()

(TypedCollection) .**\_\_len\_\_**()

(TypedCollection) .**\_\_ne\_\_**(*expr*)

(AbjadObject) .**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList) .**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList) .**\_\_setitem\_\_**(*i*, *expr*)

Change tokens in *expr* to items and set:

```

>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch('gqs,')
],
item_class=pitchtools.NamedPitch
)

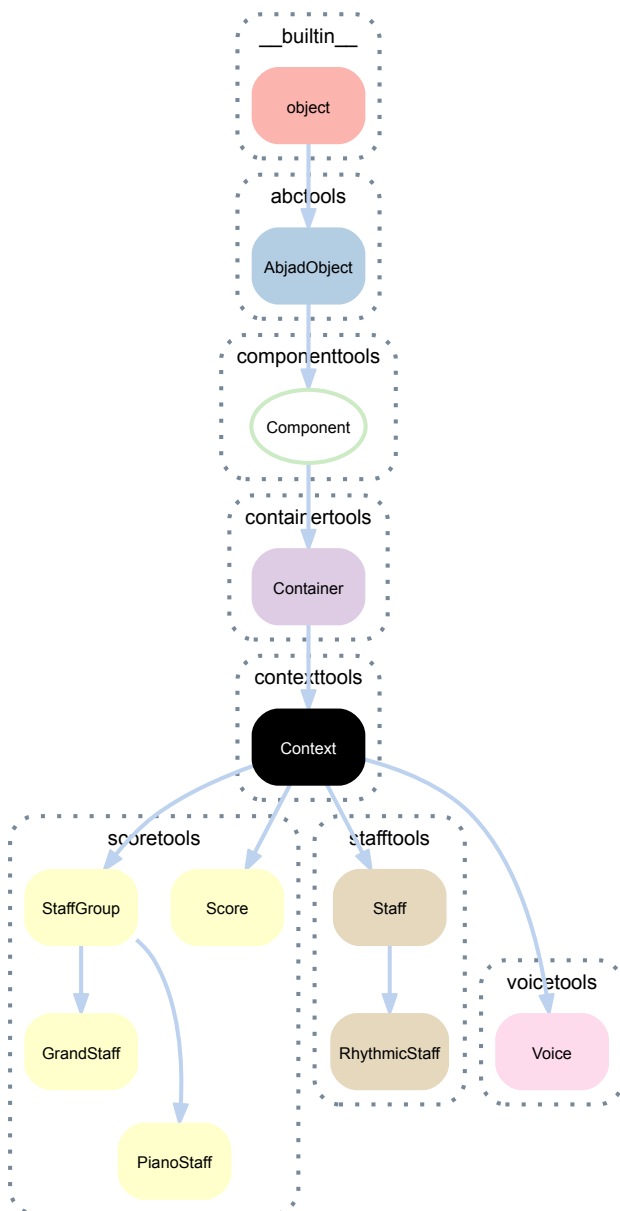
```

```

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch("f'"),
  pitchtools.NamedPitch("g'"),
  pitchtools.NamedPitch("a'"),
  pitchtools.NamedPitch("b'"),
  pitchtools.NamedPitch("c'")
],
item_class=pitchtools.NamedPitch
)

```

### 4.1.3 contexttools.Context



**class** contexttools.**Context** (*music=None, context\_name='Context', name=None*)  
 A horizontal layer of music.

```
>>> context = contexttools.Context(
...     name='MeterVoice', context_name='TimeSignatureContext')
```

```
>>> context
TimeSignatureContext-"MeterVoice"{} 
```

Returns context object.

#### Bases

- containertools.Container
- componenttools.Component
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

### Context.**engraver\_consists**

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

### Context.**engraver\_removals**

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

### Context.**is\_semantic**

### Context.**lilypond\_format**

#### (Component).**override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

#### (Component).**set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

#### (Component).**storage\_format**

Storage format of component.

Returns string.

## Read/write properties

### Context.**context\_name**

Read / write name of context as a string.

### Context.**is\_nonsemantic**

Set indicator of nonsemantic voice:

```
>>> measures = \
...   measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Returns boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interaction and other functions.

(Container) **.is\_simultaneous**  
Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

Context **.name**

Read-write name of context. Must be string or none.

## Methods

(Container) **.append**(*component*)  
Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) .**extend** (*expr*)  
 Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



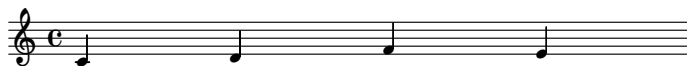
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) .**index** (*component*)  
 Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) .**insert** (*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs'e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

`(Container) .pop(i=-1)`

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```

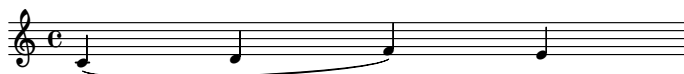


Returns component.

`(Container) .remove(component)`

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

`(Container) .reverse()`

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (*sequential=False*)  
Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)  
Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest("r8"), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords** ()  
Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container) **.\_\_contains\_\_** (*expr*)  
True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component) **.\_\_copy\_\_** (\*args)  
Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(Container) **.\_\_delitem\_\_** (*i*)  
Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Returns none.

(AbjadObject) **.\_\_eq\_\_** (*expr*)  
True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container) **.\_\_getitem\_\_** (*i*)  
Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.



(Container).**\_\_len\_\_**()  
 Number of items in container.  
 Returns nonnegative integer.

(Component).**\_\_mul\_\_**(*n*)  
 Copies component *n* times and detaches spanners.  
 Returns list of new components.

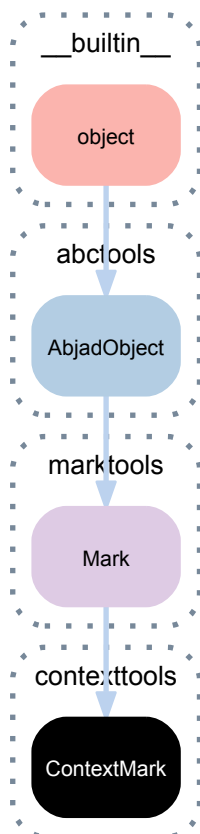
(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

Context.**\_\_repr\_\_**()

(Component).**\_\_rmul\_\_**(*n*)  
 Copies component *n* times and detach spanners.  
 Returns list of new components.

(Container).**\_\_setitem\_\_**(*i*, *expr*)  
 Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with '*expr*'. Reattach spanners to new contents. This operation always leaves score tree in tact.  
 Returns none.

#### 4.1.4 contexttools.ContextMark



**class** contexttools.**ContextMark** (*target\_context=None*)  
 Abstract class from which concrete context marks inherit.  
 Context marks are immutable.

## Bases

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ContextMark.effective_context`

Effective context of context mark.

Returns context mark or none.

`(Mark).start_component`

Start component of mark.

Returns component or none.

`(Mark).storage_format`

Storage format of mark.

Returns string.

`ContextMark.target_context`

Target context of context mark.

Returns context or none.

## Methods

`ContextMark.attach(start_component)`

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

`ContextMark.detach()`

Detaches context mark.

Returns context mark.

## Special methods

`(Mark).__call__(*args)`

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

`ContextMark.__copy__(*args)`

Copies context mark.

Returns new context mark.

`(Mark).__eq__(expr)`

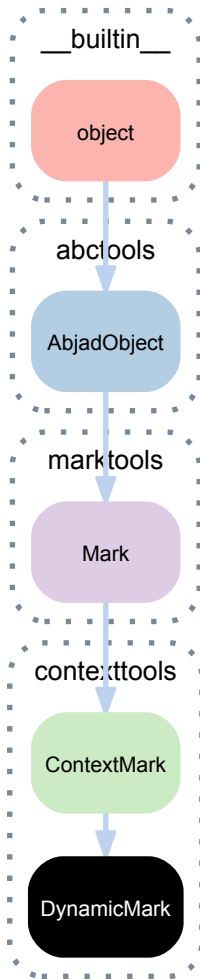
True when *expr* is the same type as self. Otherwise false.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Mark) .**\_\_repr\_\_**()  
 Interpreter representation of mark.  
 Returns string.

#### 4.1.5 contexttools.DynamicMark



**class** contexttools.**DynamicMark**(*dynamic\_name*, *target\_context=None*)  
 A dynamic mark.

**Example 1.** Initialize from dynamic name:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> contexttools.DynamicMark('f')(staff[0])
DynamicMark('f')(c'8)
```

```
>>> show(staff)
```



**Example 2.** Initialize from other dynamic mark:

```
>>> dynamic_mark_1 = contexttools.DynamicMark('f')
>>> dynamic_mark_2 = contexttools.DynamicMark(dynamic_mark_1)
```

```
>>> dynamic_mark_1
DynamicMark('f')
```

```
>>> dynamic_mark_2
DynamicMark('f')
```

Dynamic marks target the staff context by default.

### Bases

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`

Effective context of context mark.

Returns context mark or none.

`DynamicMark.lilypond_format`

LilyPond input format of dynamic mark:

```
>>> dynamic_mark = contexttools.DynamicMark('f')
>>> dynamic_mark.lilypond_format
'\\f'
```

Returns string.

`(Mark).start_component`

Start component of mark.

Returns component or none.

`(Mark).storage_format`

Storage format of mark.

Returns string.

`(ContextMark).target_context`

Target context of context mark.

Returns context or none.

### Read/write properties

`DynamicMark.dynamic_name`

Get dynamic name:

```
>>> dynamic = contexttools.DynamicMark('f')
>>> dynamic.dynamic_name
'f'
```

Set dynamic name:

```
>>> dynamic.dynamic_name = 'p'
>>> dynamic.dynamic_name
'p'
```

Returns string.

## Methods

(ContextMark) **.attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Static methods

DynamicMark.**composite\_dynamic\_name\_to\_steady\_state\_dynamic\_name**(*dynamic\_name*)

Change composite *dynamic\_name* to steady state dynamic name:

```
>>> contexttools.DynamicMark.composite_dynamic_name_to_steady_state_dynamic_name('sfp')
'p'
```

Returns string.

DynamicMark.**dynamic\_name\_to\_dynamic\_ordinal**(*dynamic\_name*)

Change *dynamic\_name* to dynamic ordinal:

```
>>> contexttools.DynamicMark.dynamic_name_to_dynamic_ordinal('fff')
4
```

Returns integer.

DynamicMark.**dynamic\_ordinal\_to\_dynamic\_name**(*dynamic\_ordinal*)

Change *dynamic\_ordinal* to dynamic name:

```
>>> contexttools.DynamicMark.dynamic_ordinal_to_dynamic_name(-5)
'pppp'
```

Returns string.

DynamicMark.**is\_dynamic\_name**(*arg*)

True when *arg* is dynamic name. False otherwise:

```
>>> contexttools.DynamicMark.is_dynamic_name('f')
True
```

Returns boolean.

## Special methods

DynamicMark.**\_\_call\_\_**(\*args)

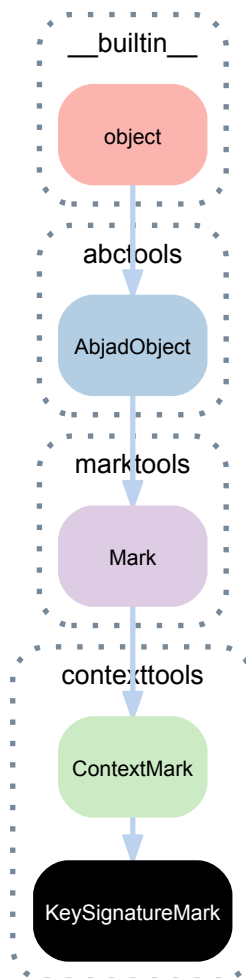
DynamicMark.**\_\_copy\_\_**(\*args)

DynamicMark.**\_\_eq\_\_**(arg)

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Mark) .**\_\_repr\_\_**()  
 Interpreter representation of mark.  
 Returns string.

#### 4.1.6 contexttools.KeySignatureMark



**class** contexttools.**KeySignatureMark**(*tonic, mode, target\_context=None*)  
 A key signature setting or key signature change.

```
>>> staff = Staff("e'8 fs'8 gs'8 a'8")
```

```
>>> contexttools.KeySignatureMark('e', 'major')(staff)
KeySignatureMark(NamedPitchClass('e'), Mode('major'))(Staff{4})
```

```
>>> show(staff)
```



Key signature marks target staff context by default.

## Bases

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`KeySignatureMark.lilypond_format`  
LilyPond format of key signature mark:

```
>>> key_signature = contexttools.KeySignatureMark('e', 'major')
>>> key_signature.lilypond_format
'\\key e \\major'
```

Returns string.

`KeySignatureMark.name`  
Name of key signature:

```
>>> key_signature = contexttools.KeySignatureMark('e', 'major')
>>> key_signature.name
'E major'
```

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`KeySignatureMark.mode`  
Get mode of key signature:

```
>>> key_signature = contexttools.KeySignatureMark('e', 'major')
>>> key_signature.mode
Mode('major')
```

Set mode of key signature:

```
>>> key_signature.mode = 'minor'
>>> key_signature.mode
Mode('minor')
```

Returns mode.

KeySignatureMark.**tonic**

Get tonic of key signature:

```
>>> key_signature = contexttools.KeySignatureMark('e', 'major')
>>> key_signature.tonic
NamedPitchClass('e')
```

Set tonic of key signature:

```
>>> key_signature.tonic = 'd'
>>> key_signature.tonic
NamedPitchClass('d')
```

Returns named pitch.

## Methods

(ContextMark) **.attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

KeySignatureMark.**\_\_copy\_\_**(\*args)

KeySignatureMark.**\_\_eq\_\_**(arg)

(AbjadObject) **.\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Mark) **.\_\_repr\_\_**()

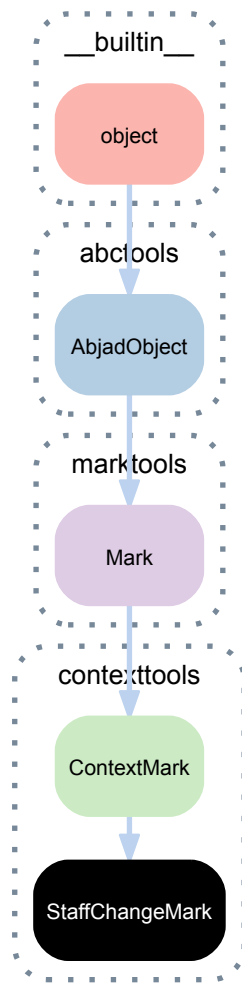
Interpreter representation of mark.

Returns string.

KeySignatureMark.**\_\_str\_\_**()



### 4.1.7 contexttools.StaffChangeMark



**class** contexttools.StaffChangeMark (staff=None, target\_context=None)  
A staff change.

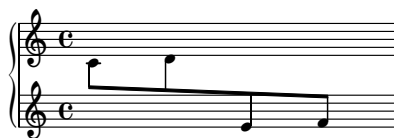
```
>>> piano_staff = scoretools.PianoStaff([])
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> piano_staff.extend([rh_staff, lh_staff])
```

```
>>> show(piano_staff)
```



```
>>> contexttools.StaffChangeMark(lh_staff)(rh_staff[2])
StaffChangeMark(Staff-"LHStaff"{1})(e'8)
```

```
>>> show(piano_staff)
```



Staff change marks target staff context by default.

## Bases

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**

Effective context of context mark.

Returns context mark or none.

StaffChangeMark **.lilypond\_format**

LilyPond format of staff change mark:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff.name = 'RHStaff'
>>> staff_change = contexttools.StaffChangeMark(staff)
>>> staff_change.lilypond_format
'\change Staff = RHStaff'
```

Returns string.

(Mark) **.start\_component**

Start component of mark.

Returns component or none.

(Mark) **.storage\_format**

Storage format of mark.

Returns string.

(ContextMark) **.target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

StaffChangeMark **.staff**

Get staff of staff change mark:

```
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> staff_change = contexttools.StaffChangeMark(rh_staff)
>>> staff_change.staff
Staff-"RHStaff"{4}
```

Set staff of staff change mark:

```
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> staff_change.staff = lh_staff
>>> staff_change.staff
Staff-"LHStaff"{1}
```

Returns staff.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

StaffChangeMark .**\_\_copy\_\_**(\*args)

StaffChangeMark .**\_\_eq\_\_**(arg)

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

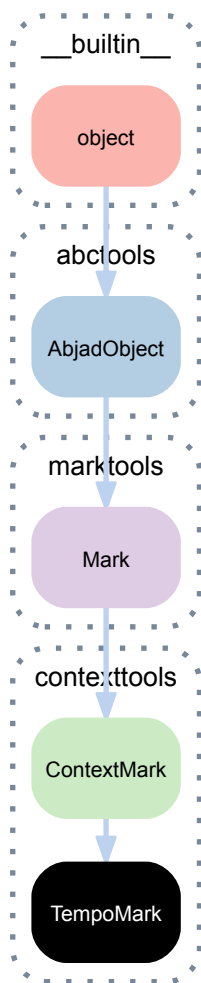
Returns boolean.

(Mark) .**\_\_repr\_\_**()

Interpreter representation of mark.

Returns string.

### 4.1.8 contexttools.TempoMark



**class contexttools.TempoMark** (\*args, \*\*kwargs)  
A tempo indication.

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
```

```
>>> contexttools.TempoMark(Duration(1, 8), 52)(staff[0])
TempoMark(Duration(1, 8), 52)(c'8)
```

```
>>> show(score)
```



Tempo marks target **score** context by default.

Initialization allows many different types of input argument structure.

#### Bases

- contexttools.ContextMark
- marktools.Mark
- abctools.AbjadObject

- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**

Effective context of context mark.

Returns context mark or none.

TempoMark **.is\_imprecise**

True if tempo mark is entirely textual, or if tempo mark's `units_per_minute` is a range:

```
>>> contexttools.TempoMark(Duration(1, 4), 60).is_imprecise
False
>>> contexttools.TempoMark('Langsam', 4, 60).is_imprecise
False
>>> contexttools.TempoMark('Langsam').is_imprecise
True
>>> contexttools.TempoMark('Langsam', 4, (35, 50)).is_imprecise
True
>>> contexttools.TempoMark(Duration(1, 4), (35, 50)).is_imprecise
True
```

Returns boolean.

TempoMark **.lilypond\_format**

LilyPond format of tempo mark:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo.lilypond_format
'\\tempo 8=52'
```

```
>>> tempo.textual_indication = 'Gingerly'
>>> tempo.lilypond_format
'\\tempo Gingerly 8=52'
```

```
>>> tempo.units_per_minute = (52, 56)
>>> tempo.lilypond_format
'\\tempo Gingerly 8=52-56'
```

Returns string.

TempoMark **.quarters\_per\_minute**

Quarters per minute of tempo mark:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo.quarters_per_minute
Fraction(104, 1)
```

Returns fraction.

Or tuple if tempo mark `units_per_minute` is a range.

Or none if tempo mark is imprecise.

(Mark) **.start\_component**

Start component of mark.

Returns component or none.

(Mark) **.storage\_format**

Storage format of mark.

Returns string.

(ContextMark) **.target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

`TempoMark.duration`

Get duration of tempo mark:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo.duration
Duration(1, 8)
```

Set duration of tempo mark:

```
>>> tempo.duration = Duration(1, 4)
>>> tempo.duration
Duration(1, 4)
```

Returns duration, or None if tempo mark is imprecise.

`TempoMark.textual_indication`

Get textual indication of tempo mark:

```
>>> tempo = contexttools.TempoMark(
...     'Langsam', Duration(1, 8), 52)
>>> tempo.textual_indication
'Langsam'
```

Returns string or None.

`TempoMark.units_per_minute`

Get units per minute of tempo mark:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo.units_per_minute
52
```

Set units per minute of tempo mark:

```
>>> tempo.units_per_minute = 56
>>> tempo.units_per_minute
56
```

Returns number.

## Methods

`(ContextMark).attach(start_component)`

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

`(ContextMark).detach()`

Detaches context mark.

Returns context mark.

`TempoMark.duration_to_milliseconds(duration)`

Returns the millisecond value of *duration* under a given tempo:

```
>>> duration = (1, 4)
>>> tempo = contexttools.TempoMark((1, 4), 60)
>>> tempo.duration_to_milliseconds(duration)
Duration(1000, 1)
```

Returns duration.

`TempoMark.is_tempo_mark_token(expr)`

True when *expr* has the form of a tempo mark initializer:

```
>>> tempo_mark = contexttools.TempoMark(Duration(1, 4), 72)
>>> tempo_mark.is_tempo_mark_token((Duration(1, 4), 84))
True
```

Otherwise false:

```
>>> tempo_mark.is_tempo_mark_token(84)
False
```

Returns boolean.

## Special methods

`TempoMark.__add__(expr)`

(Mark) `.__call__(*args)`

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

`TempoMark.__copy__(*args)`

`TempoMark.__div__(expr)`

`TempoMark.__eq__(expr)`

`TempoMark.__mul__(multiplier)`

(AbjadObject) `.__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

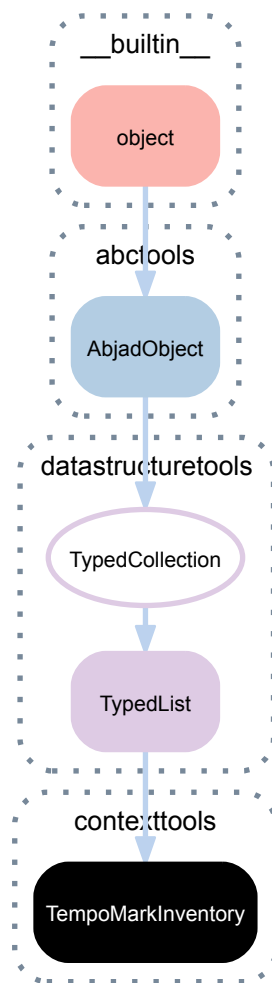
(Mark) `.__repr__()`

Interpreter representation of mark.

Returns string.

`TempoMark.__sub__(expr)`

### 4.1.9 contexttools.TempoMarkInventory



**class** contexttools.**TempoMarkInventory** (*tokens=None, item\_class=None, name=None*)  
 An ordered list of tempo marks.

```
>>> inventory = contexttools.TempoMarkInventory([
...     ('Andante', Duration(1, 8), 72),
...     ('Allegro', Duration(1, 8), 84)])
```

```
>>> for tempo_mark in inventory:
...     tempo_mark
...
TempoMark('Andante', Duration(1, 8), 72)
TempoMark('Allegro', Duration(1, 8), 84)
```

Tempo mark inventories implement list interface and are mutable.

#### Bases

- datastructuretools.TypedList
- datastructuretools.TypedCollection
- abctools.AbjadObject
- \_\_builtin\_\_.object



## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedList).append(token)`  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

`(TypedList).count(token)`  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)
3
```

Returns count.

`(TypedList).extend(tokens)`  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

`(TypedList).index(token)`  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('c'qf', "as'", 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

(TypedList) **.insert** (*i*, *token*)  
 Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('1', 2, 4.3))
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None*, *item\_class=None*, *name=None*)

(TypedList) **.pop** (*i=-1*)  
 Aliases list.pop().

(TypedList) **.remove** (*token*)  
 Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()  
 Aliases list.reverse().

(TypedList) **.sort** (*cmp=None*, *key=None*, *reverse=False*)  
 Aliases list.sort().

## Special methods

(TypedCollection) **.\_\_contains\_\_** (*token*)

(TypedList) **.\_\_delitem\_\_** (*i*)  
 Aliases list.\_\_delitem\_\_().

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedList) **.\_\_getitem\_\_** (*i*)  
 Aliases list.\_\_getitem\_\_().

(TypedList) **.\_\_iadd\_\_** (*expr*)  
 Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList (
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList ([
    contexttools.DynamicMark (
```

```

        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'fff',
        target_context=stafftools.Staff
    )
],
item_class=contexttools.DynamicMark
)

```

Returns collection.

(TypedCollection) .**\_\_iter\_\_**()

(TypedCollection) .**\_\_len\_\_**()

(TypedCollection) .**\_\_ne\_\_**(*expr*)

(AbjadObject) .**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList) .**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList) .**\_\_setitem\_\_**(*i*, *expr*)

Change tokens in *expr* to items and set:

```

>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch('gqs,')
],
item_class=pitchtools.NamedPitch
)

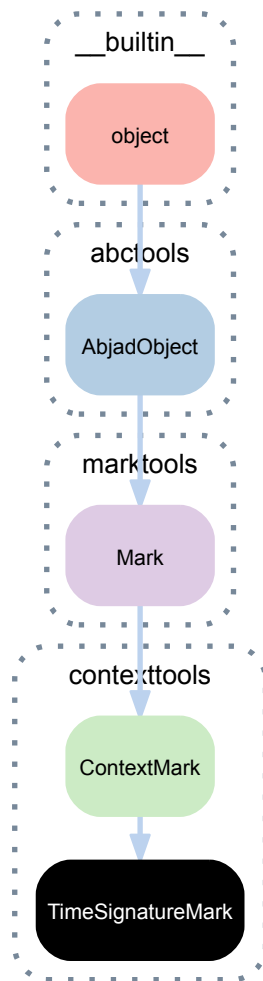
```

```

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch("f'"),
  pitchtools.NamedPitch("g'"),
  pitchtools.NamedPitch("a'"),
  pitchtools.NamedPitch("b'"),
  pitchtools.NamedPitch("c'")
],
item_class=pitchtools.NamedPitch
)

```

#### 4.1.10 contexttools.TimeSignatureMark



**class** contexttools.**TimeSignatureMark** (\*args, \*\*kwargs)  
A time signature.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> contexttools.TimeSignatureMark((4, 8))(staff[0])
TimeSignatureMark((4, 8))(c'8)
```

```
>>> show(staff)
```



Abjad time signature marks target **staff context** by default.

Initialize time signature marks to **score context** like this:

```
>>> contexttools.TimeSignatureMark((4, 8), target_context=Score)
TimeSignatureMark((4, 8), target_context=Score)
```

#### Bases

- contexttools.ContextMark
- marktools.Mark
- abctools.AbjadObject

- `__builtin__.object`

## Read-only properties

`TimeSignatureMark.duration`

Time signature mark duration:

```
>>> contexttools.TimeSignatureMark((3, 8)).duration
Duration(3, 8)
```

Returns duration.

`TimeSignatureMark.effective_context`

Time signature mark effective context.

Returns none when time signature mark is not yet attached:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
```

```
>>> time_signature.effective_context is None
True
```

Returns context when time signature mark is attached:

```
>>> staff = Staff()
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8))(Staff{})
```

```
>>> time_signature.effective_context
Staff{}
```

Returns context or none.

`TimeSignatureMark.has_non_power_of_two_denominator`

True when time signature mark has non-power-of-two denominator:

```
>>> time_signature = contexttools.TimeSignatureMark((7, 12))
>>> time_signature.has_non_power_of_two_denominator
True
```

Otherwise false:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> time_signature.has_non_power_of_two_denominator
False
```

Returns boolean.

`TimeSignatureMark.implied_prolation`

Time signature mark implied prolation.

**Example 1.** Implied prolation of time signature with power-of-two denominator:

```
>>> contexttools.TimeSignatureMark((3, 8)).implied_prolation
Multiplier(1, 1)
```

**Example 2.** Implied prolation of time signature with non-power-of-two denominator:

```
>>> contexttools.TimeSignatureMark((7, 12)).implied_prolation
Multiplier(2, 3)
```

Returns multiplier.

`TimeSignatureMark.lilypond_format`

Time signature mark LilyPond format:

```
>>> contexttools.TimeSignatureMark((3, 8)).lilypond_format
'\time 3/8'
```

Returns string.

`TimeSignatureMark.pair`

Time signature numerator / denominator pair:

```
>>> contexttools.TimeSignatureMark((3, 8)).pair
(3, 8)
```

Returns pair.

`TimeSignatureMark.start_component`

Time signature mark start component.

Returns none when time signature mark is not yet attached:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> time_signature.start_component is None
True
```

Returns component when time signature mark is attached:

```
>>> staff = Staff()
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8))(Staff{})
```

```
>>> time_signature.start_component
Staff{}
```

Returns component or none.

`TimeSignatureMark.storage_format`

Time signature mark storage format:

```
>>> print contexttools.TimeSignatureMark((3, 8)).storage_format
contexttools.TimeSignatureMark(
  (3, 8)
)
```

Returns string.

`TimeSignatureMark.target_context`

Time signature mark target context:

```
>>> contexttools.TimeSignatureMark((3, 8)).target_context
<class 'abjad.tools.stafftools.Staff.Staff.Staff'>
```

Time signature marks target the staff context by default.

This can be changed at initialization.

Returns class.

## Read/write properties

`TimeSignatureMark.denominator`

Get denominator of time signature mark:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
```

```
>>> time_signature.denominator
8
```

Set denominator of time signature mark:

```
>>> time_signature.denominator = 16
```

```
>>> time_signature.denominator
16
```

Returns integer.

`TimeSignatureMark.numerator`

Get numerator of time signature mark:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> time_signature.numerator
3
```

Set numerator of time signature mark:

```
>>> time_signature.numerator = 4
>>> time_signature.numerator
4
```

Set integer.

`TimeSignatureMark.partial`

Get partial measure pick-up of time signature mark:

```
>>> time_signature = contexttools.TimeSignatureMark(
...     (3, 8), partial=Duration(1, 8))
>>> time_signature.partial
Duration(1, 8)
```

Set partial measure pick-up of time signature mark:

```
>>> time_signature.partial = Duration(1, 4)
>>> time_signature.partial
Duration(1, 4)
```

Set duration or none.

## Methods

`TimeSignatureMark.attach(start_component)`

Attach time signature mark to *start\_component*:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> staff = Staff()
```

```
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8))(Staff{})
```

Returns time signature mark.

`TimeSignatureMark.detach()`

Detaches time signature mark.

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> staff = Staff()
```

```
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8))(Staff{})
```

```
>>> time_signature.detach()
TimeSignatureMark((3, 8))
```

Returns time signature mark.

`TimeSignatureMark.with_power_of_two_denominator(contents_multiplier=Multiplier(1, 1))`

Create new time signature equivalent to current time signature with power-of-two denominator.

```
>>> time_signature = contexttools.TimeSignatureMark((3, 12))
```

```
>>> time_signature.with_power_of_two_denominator()
TimeSignatureMark((2, 8))
```

Returns new time signature mark.

### Special methods

(Mark) .**\_\_call\_\_**(*\*args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

TimeSignatureMark.**\_\_copy\_\_**(*\*args*)

TimeSignatureMark.**\_\_eq\_\_**(*arg*)

TimeSignatureMark.**\_\_ge\_\_**(*arg*)

TimeSignatureMark.**\_\_gt\_\_**(*arg*)

TimeSignatureMark.**\_\_le\_\_**(*arg*)

TimeSignatureMark.**\_\_lt\_\_**(*arg*)

TimeSignatureMark.**\_\_ne\_\_**(*arg*)

TimeSignatureMark.**\_\_nonzero\_\_**()

TimeSignatureMark.**\_\_repr\_\_**()

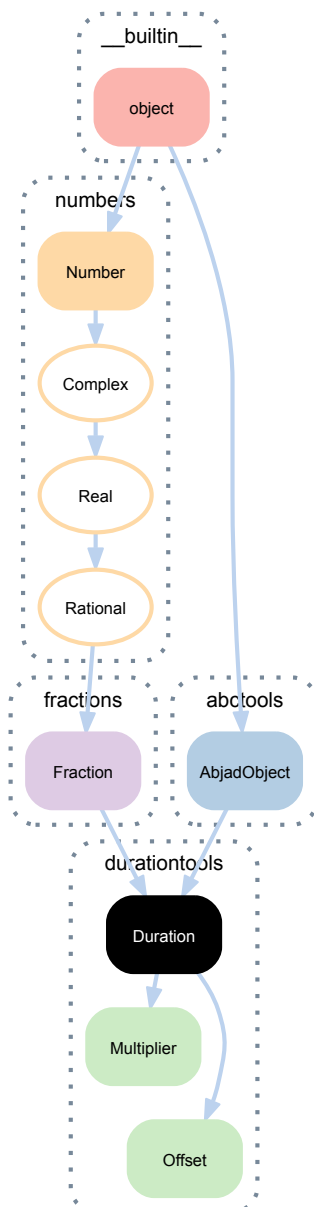
TimeSignatureMark.**\_\_str\_\_**()



# DURATIONTOOLS

## 5.1 Concrete classes

### 5.1.1 durationtools.Duration



**class** `durationtools.Duration`

A musical duration.

**Initializer examples.**

Initialize from integer numerator:

```
>>> Duration(3)
Duration(3, 1)
```

Initialize from integer numerator and denominator:

```
>>> Duration(3, 16)
Duration(3, 16)
```

Initialize from integer-equivalent numeric numerator:

```
>>> Duration(3.0)
Duration(3, 1)
```

Initialize from integer-equivalent numeric numerator and denominator:

```
>>> Duration(3.0, 16)
Duration(3, 16)
```

Initialize from integer-equivalent singleton:

```
>>> Duration((3,))
Duration(3, 1)
```

Initialize from integer-equivalent pair:

```
>>> Duration((3, 16))
Duration(3, 16)
```

Initialize from other duration:

```
>>> Duration(Duration(3, 16))
Duration(3, 16)
```

Initialize from fraction:

```
>>> Duration(Fraction(3, 16))
Duration(3, 16)
```

Initialize from solidus string:

```
>>> Duration('3/16')
Duration(3, 16)
```

Initialize from nonreduced fraction:

```
>>> Duration(mathtools.NonreducedFraction(3, 16))
Duration(3, 16)
```

Durations inherit from built-in fraction:

```
>>> isinstance(Duration(3, 16), Fraction)
True
```

Durations are numeric:

```
>>> import numbers
```

```
>>> isinstance(Duration(3, 16), numbers.Number)
True
```

## Bases

- `abctools.AbjadObject`
- `fractions.Fraction`
- `numbers.Rational`
- `numbers.Real`
- `numbers.Complex`
- `numbers.Number`
- `__builtin__.object`

## Read-only properties

`(Fraction).denominator`

`Duration.dot_count`

Positive integer number of dots required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         print '{}\t{}'.format(
...             duration.with_denominator(16), duration.dot_count)
...     except AssignabilityError:
...         print '{}\t{}'.format(
...             duration.with_denominator(16), '--')
...
1/16    0
2/16    0
3/16    1
4/16    0
5/16    --
6/16    1
7/16    2
8/16    0
9/16    --
10/16   --
11/16   --
12/16    1
13/16   --
14/16    2
15/16    3
16/16    0
```

Returns positive integer.

Raise assignability error when duration is not assignable.

`Duration.equal_or_greater_assignable`

Equal or greater assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    3/8
6/16    3/8
7/16    7/16
8/16    1/2
9/16    3/4
```

```

10/16  3/4
11/16  3/4
12/16  3/4
13/16  7/8
14/16  7/8
15/16  15/16
16/16  1

```

Returns new duration.

**Duration.equal\_or\_greater\_power\_of\_two**  
 Equal or greater power of 2:

```

>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16  1/16
2/16  1/8
3/16  1/4
4/16  1/4
5/16  1/2
6/16  1/2
7/16  1/2
8/16  1/2
9/16  1
10/16 1
11/16 1
12/16 1
13/16 1
14/16 1
15/16 1
16/16 1

```

Returns new duration.

**Duration.equal\_or\_lesser\_assignable**  
 Equal or lesser assignable:

```

>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16  1/16
2/16  1/8
3/16  3/16
4/16  1/4
5/16  1/4
6/16  3/8
7/16  7/16
8/16  1/2
9/16  1/2
10/16 1/2
11/16 1/2
12/16 3/4
13/16 3/4
14/16 7/8
15/16 15/16
16/16 1

```

Returns new duration.

**Duration.equal\_or\_lesser\_power\_of\_two**  
 Equal or lesser power of 2:

```

>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_power_of_two

```

```

...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    1/8
4/16    1/4
5/16    1/4
6/16    1/4
7/16    1/4
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   1/2
13/16   1/2
14/16   1/2
15/16   1/2
16/16   1

```

Returns new duration.

#### `Duration.flag_count`

Nonnegative integer number of flags required to notate duration:

```

>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     print '{}\t{}'.format(
...         duration.with_denominator(64), duration.flag_count)
...
1/64    4
2/64    3
3/64    3
4/64    2
5/64    2
6/64    2
7/64    2
8/64    1
9/64    1
10/64   1
11/64   1
12/64   1
13/64   1
14/64   1
15/64   1
16/64   0

```

Returns nonnegative integer.

#### `Duration.has_power_of_two_denominator`

True when duration is an integer power of 2. Otherwise false:

```

>>> for n in range(1, 16 + 1):
...     duration = Duration(1, n)
...     print '{}\t{}'.format(duration,
...         duration.has_power_of_two_denominator)
...
1      True
1/2    True
1/3    False
1/4    True
1/5    False
1/6    False
1/7    False
1/8    True
1/9    False
1/10   False
1/11   False
1/12   False
1/13   False
1/14   False
1/15   False

```

1/16      True

Returns boolean.

(Real).**.imag**

Real numbers have no imaginary component.

Duration.**implied\_prolation**

Implied prololation of multiplier:

```
>>> for denominator in range(1, 16 + 1):
...     multiplier = Multiplier(1, denominator)
...     print '{}\t{}'.format(multiplier,
...         multiplier.implied_prolation)
...
1          1
1/2        1
1/3        2/3
1/4        1
1/5        4/5
1/6        2/3
1/7        4/7
1/8        1
1/9        8/9
1/10       4/5
1/11       8/11
1/12       2/3
1/13       8/13
1/14       4/7
1/15       8/15
1/16       1
```

Returns new multiplier.

Duration.**is\_assignable**

True when assignable. Otherwise false:

```
>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     print '{}\t{}'.format(duration.with_denominator(16),
...         duration.is_assignable)
...
0/16      False
1/16      True
2/16      True
3/16      True
4/16      True
5/16      False
6/16      True
7/16      True
8/16      True
9/16      False
10/16     False
11/16     False
12/16     True
13/16     False
14/16     True
15/16     True
16/16     True
```

Returns boolean.

Duration.**lilypond\_duration\_string**

LilyPond duration string of assignable duration.

```
>>> Duration(3, 16).lilypond_duration_string
'8.'
```

Returns string.

Raise assignability error when duration is not assignable.

(Fraction).**numerator**

Duration.**pair**

Pair of duration numerator and denominator:

```
>>> duration = Duration(3, 16)
```

```
>>> duration.pair
(3, 16)
```

Returns integer pair.

Duration.**prolation\_string**

Prolation string:

```
>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     rational = generator.next()
...     duration = Duration(rational)
...     print '{}\t{}'.format(duration, duration.prolation_string)
...
1      1:1
2      1:2
1/2    2:1
1/3    3:1
3      1:3
4      1:4
3/2    2:3
2/3    3:2
1/4    4:1
1/5    5:1
5      1:5
6      1:6
5/2    2:5
4/3    3:4
3/4    4:3
2/5    5:2
```

Returns string.

(Real).**real**

Real numbers are their real component.

Duration.**reciprocal**

Reciprocal of duration.

Returns newly constructed duration.

Duration.**storage\_format**

Storage format of duration.

Returns string.

## Methods

(Real).**conjugate()**

Conjugate is a no-op for Reals.

(Fraction).**limit\_denominator** (*max\_denominator=1000000*)

Closest Fraction to self with denominator at most *max\_denominator*.

```
>>> Fraction('3.141592653589793').limit_denominator(10)
Fraction(22, 7)
>>> Fraction('3.141592653589793').limit_denominator(100)
Fraction(311, 99)
>>> Fraction(4321, 8765).limit_denominator(10000)
Fraction(4321, 8765)
```

`Duration.to_clock_string` (*escape\_ticks=False*)  
 Change duration to clock string.

**Example 1.** Change numeric *seconds* to clock string:

```
>>> duration = Duration(117)
>>> duration.to_clock_string()
'1\57'
```

**Example 2.** Change numeric *seconds* to escaped clock string:

```
>>> note = Note("c'4")
>>> clock_string = duration.to_clock_string(escape_ticks=True)
```

```
>>> markuptools.Markup('%s' % clock_string, Up)(note)
Markup(('1\57\'',), direction=Up)(c'4)
```

Returns string.

`Duration.with_denominator` (*denominator*)  
 Duration with *denominator*:

```
>>> duration = Duration(1, 4)

>>> for denominator in (4, 8, 16, 32):
...     print duration.with_denominator(denominator)
...
1/4
2/8
4/16
8/32
```

Returns new duration.

## Class methods

`(Fraction).from_decimal` (*dec*)  
 Converts a finite Decimal instance to a rational number, exactly.

`(Fraction).from_float` (*f*)  
 Converts a finite float to a rational number, exactly.  
 Beware that `Fraction.from_float(0.3) != Fraction(3, 10)`.

## Static methods

`Duration.durations_to_nonreduced_fractions_with_common_denominator` (*durations*)  
 Change *durations* to nonreduced fractions with least common denominator:

```
>>> durations = [Duration(2, 4), 3, (5, 16)]
>>> for x in Duration.durations_to_nonreduced_fractions_with_common_denominator(
...     durations):
...     x
...
NonreducedFraction(8, 16)
NonreducedFraction(48, 16)
NonreducedFraction(5, 16)
```

Returns new object of *durations* type.

`Duration.from_lilypond_duration_string` (*lilypond\_duration\_string*)  
 Initialize from LilyPond duration string:

```
>>> Duration.from_lilypond_duration_string('8.')
Duration(3, 16)
```

Returns duration.



`Duration.is_token(expr)`

True if *expr* correctly initializes a duration. Otherwise false.

```
>>> Duration.is_token('8.')
True
```

Returns boolean.

`Duration.yield_durations(unique=False)`

Yield durations.

**Example 1.** Yield all positive durations in Cantor diagonalized order:

```
>>> generator = Duration.yield_durations()
>>> for n in range(16):
...     generator.next()
...
Duration(1, 1)
Duration(2, 1)
Duration(1, 2)
Duration(1, 3)
Duration(1, 1)
Duration(3, 1)
Duration(4, 1)
Duration(3, 2)
Duration(2, 3)
Duration(1, 4)
Duration(1, 5)
Duration(1, 2)
Duration(1, 1)
Duration(2, 1)
Duration(5, 1)
Duration(6, 1)
```

**Example 2.** Yield all positive durations in Cantor diagonalized order uniquely:

```
>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     generator.next()
...
Duration(1, 1)
Duration(2, 1)
Duration(1, 2)
Duration(1, 3)
Duration(3, 1)
Duration(4, 1)
Duration(3, 2)
Duration(2, 3)
Duration(1, 4)
Duration(1, 5)
Duration(5, 1)
Duration(6, 1)
Duration(5, 2)
Duration(4, 3)
Duration(3, 4)
Duration(2, 5)
```

Returns generator.

## Special methods

`Duration.__abs__(*args)`

`Duration.__add__(*args)`

`(Real).__complex__()`  
`complex(self) == complex(float(self), 0)`

`(Fraction).__copy__()`

`(Fraction).__deepcopy__(memo)`

Duration.\_\_div\_\_(\*args)

Duration.\_\_divmod\_\_(\*args)

Duration.\_\_eq\_\_(arg)

(Rational).\_\_float\_\_()

float(self) = self.numerator / self.denominator

It's important that this conversion use the integer's "true" division rather than casting one side to float before dividing so that ratios of huge integers convert without overflowing.

(Fraction).\_\_floordiv\_\_(a, b)

a // b

Duration.\_\_ge\_\_(arg)

Duration.\_\_gt\_\_(arg)

(Fraction).\_\_hash\_\_()

hash(self)

Tricky because values that are exactly representable as a float must have the same hash as that float.

Duration.\_\_le\_\_(arg)

Duration.\_\_lt\_\_(arg)

Duration.\_\_mod\_\_(\*args)

Duration.\_\_mul\_\_(\*args)

Duration.\_\_ne\_\_(arg)

Duration.\_\_neg\_\_(\*args)

Duration.\_\_new\_\_(\*args)

(Fraction).\_\_nonzero\_\_(a)

a != 0

Duration.\_\_pos\_\_(\*args)

Duration.\_\_pow\_\_(\*args)

Duration.\_\_radd\_\_(\*args)

Duration.\_\_rdiv\_\_(\*args)

Duration.\_\_rdivmod\_\_(\*args)

Duration.\_\_repr\_\_()

(Fraction).\_\_rfloordiv\_\_(b, a)

a // b

Duration.\_\_rmod\_\_(\*args)

Duration.\_\_rmul\_\_(\*args)

Duration.\_\_rpow\_\_(\*args)

Duration.\_\_rsub\_\_(\*args)

Duration.\_\_rtruediv\_\_(\*args)

(Fraction).\_\_str\_\_()

str(self)

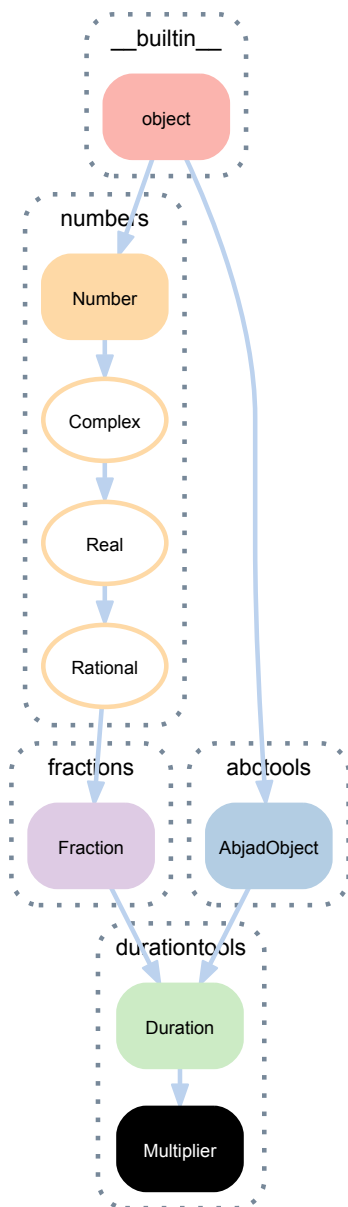
Duration.\_\_sub\_\_(\*args)

Duration.\_\_truediv\_\_(\*args)

(Fraction).\_\_trunc\_\_(a)

trunc(a)

### 5.1.2 durationtools.Multiplier



**class** `durationtools.Multiplier`  
A multiplier.

```
>>> Multiplier(2, 3)
Multiplier(2, 3)
```

#### Bases

- `durationtools.Duration`
- `abctools.AbjadObject`
- `fractions.Fraction`
- `numbers.Rational`
- `numbers.Real`
- `numbers.Complex`

- `numbers.Number`
- `__builtin__.object`

## Read-only properties

`(Fraction).denominator`

`(Duration).dot_count`

Positive integer number of dots required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         print '{}\t{}'.format(
...             duration.with_denominator(16), duration.dot_count)
...     except AssignabilityError:
...         print '{}\t{}'.format(
...             duration.with_denominator(16), '--')
...
1/16    0
2/16    0
3/16    1
4/16    0
5/16    --
6/16    1
7/16    2
8/16    0
9/16    --
10/16   --
11/16   --
12/16    1
13/16   --
14/16    2
15/16    3
16/16    0
```

Returns positive integer.

Raise assignability error when duration is not assignable.

`(Duration).equal_or_greater_assignable`

Equal or greater assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    3/8
6/16    3/8
7/16    7/16
8/16    1/2
9/16    3/4
10/16   3/4
11/16   3/4
12/16   3/4
13/16   7/8
14/16   7/8
15/16   15/16
16/16   1
```

Returns new duration.

`(Duration).equal_or_greater_power_of_two`

Equal or greater power of 2:

```

>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    1/4
4/16    1/4
5/16    1/2
6/16    1/2
7/16    1/2
8/16    1/2
9/16    1
10/16   1
11/16   1
12/16   1
13/16   1
14/16   1
15/16   1
16/16   1

```

Returns new duration.

(Duration).**equal\_or\_lesser\_assignable**

Equal or lesser assignable:

```

>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    1/4
6/16    3/8
7/16    7/16
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   3/4
13/16   3/4
14/16   7/8
15/16   15/16
16/16   1

```

Returns new duration.

(Duration).**equal\_or\_lesser\_power\_of\_two**

Equal or lesser power of 2:

```

>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_power_of_two
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    1/8
4/16    1/4
5/16    1/4
6/16    1/4
7/16    1/4
8/16    1/2
9/16    1/2
10/16   1/2

```

```

11/16  1/2
12/16  1/2
13/16  1/2
14/16  1/2
15/16  1/2
16/16  1

```

Returns new duration.

(Duration).**flag\_count**

Nonnegative integer number of flags required to notate duration:

```

>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     print '{}\t{}'.format(
...         duration.with_denominator(64), duration.flag_count)
...
1/64    4
2/64    3
3/64    3
4/64    2
5/64    2
6/64    2
7/64    2
8/64    1
9/64    1
10/64   1
11/64   1
12/64   1
13/64   1
14/64   1
15/64   1
16/64   0

```

Returns nonnegative integer.

(Duration).**has\_power\_of\_two\_denominator**

True when duration is an integer power of 2. Otherwise false:

```

>>> for n in range(1, 16 + 1):
...     duration = Duration(1, n)
...     print '{}\t{}'.format(duration,
...         duration.has_power_of_two_denominator)
...
1      True
1/2    True
1/3    False
1/4    True
1/5    False
1/6    False
1/7    False
1/8    True
1/9    False
1/10   False
1/11   False
1/12   False
1/13   False
1/14   False
1/15   False
1/16   True

```

Returns boolean.

(Real).**imag**

Real numbers have no imaginary component.

(Duration).**implied\_prolation**

Implied prolotion of multiplier:

```

>>> for denominator in range(1, 16 + 1):
...     multiplier = Multiplier(1, denominator)

```

```

...     print '{}\t{}'.format(multiplier,
...                             multiplier.implied_prolation)
...
1         1
1/2       1
1/3       2/3
1/4       1
1/5       4/5
1/6       2/3
1/7       4/7
1/8       1
1/9       8/9
1/10      4/5
1/11      8/11
1/12      2/3
1/13      8/13
1/14      4/7
1/15      8/15
1/16      1

```

Returns new multiplier.

(Duration).**.is\_assignable**

True when assignable. Otherwise false:

```

>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     print '{}\t{}'.format(duration.with_denominator(16),
...                             duration.is_assignable)
...
0/16      False
1/16      True
2/16      True
3/16      True
4/16      True
5/16      False
6/16      True
7/16      True
8/16      True
9/16      False
10/16     False
11/16     False
12/16     True
13/16     False
14/16     True
15/16     True
16/16     True

```

Returns boolean.

Multiplier.**.is\_proper\_tuplet\_multiplier**

True when multiplier is greater than 1/2 and less than 2. Otherwise false:

```

>>> Multiplier(3, 2).is_proper_tuplet_multiplier
True

```

Returns boolean.

(Duration).**.lilypond\_duration\_string**

LilyPond duration string of assignable duration.

```

>>> Duration(3, 16).lilypond_duration_string
'8.'

```

Returns string.

Raise assignability error when duration is not assignable.

(Fraction).**.numerator**

(Duration).**.pair**

Pair of duration numerator and denominator:

```
>>> duration = Duration(3, 16)
```

```
>>> duration.pair
(3, 16)
```

Returns integer pair.

(Duration).**.prolation\_string**

Prolation string:

```
>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     rational = generator.next()
...     duration = Duration(rational)
...     print '{}\t{}'.format(duration, duration.prolation_string)
...
1      1:1
2      1:2
1/2    2:1
1/3    3:1
3      1:3
4      1:4
3/2    2:3
2/3    3:2
1/4    4:1
1/5    5:1
5      1:5
6      1:6
5/2    2:5
4/3    3:4
3/4    4:3
2/5    5:2
```

Returns string.

(Real).**.real**

Real numbers are their real component.

(Duration).**.reciprocal**

Reciprocal of duration.

Returns newly constructed duration.

(Duration).**.storage\_format**

Storage format of duration.

Returns string.

## Methods

(Real).**.conjugate()**

Conjugate is a no-op for Reals.

(Fraction).**.limit\_denominator** (*max\_denominator=1000000*)

Closest Fraction to self with denominator at most *max\_denominator*.

```
>>> Fraction('3.141592653589793').limit_denominator(10)
Fraction(22, 7)
>>> Fraction('3.141592653589793').limit_denominator(100)
Fraction(311, 99)
>>> Fraction(4321, 8765).limit_denominator(10000)
Fraction(4321, 8765)
```

(Duration).**.to\_clock\_string** (*escape\_ticks=False*)

Change duration to clock string.

**Example 1.** Change numeric *seconds* to clock string:



```
>>> duration = Duration(117)
>>> duration.to_clock_string()
'1\57''
```

**Example 2.** Change numeric *seconds* to escaped clock string:

```
>>> note = Note("c'4")
>>> clock_string = duration.to_clock_string(escape_ticks=True)
```

```
>>> markuptools.Markup('"%s"' % clock_string, Up)(note)
Markup(('1\57\'\'',), direction=Up)(c'4)
```

Returns string.

(Duration).**with\_denominator**(*denominator*)  
Duration with *denominator*:

```
>>> duration = Duration(1, 4)
```

```
>>> for denominator in (4, 8, 16, 32):
...     print duration.with_denominator(denominator)
...
1/4
2/8
4/16
8/32
```

Returns new duration.

## Class methods

(Fraction).**from\_decimal**(*dec*)  
Converts a finite Decimal instance to a rational number, exactly.

(Fraction).**from\_float**(*f*)  
Converts a finite float to a rational number, exactly.

Beware that Fraction.from\_float(0.3) != Fraction(3, 10).

## Static methods

(Duration).**durations\_to\_nonreduced\_fractions\_with\_common\_denominator**(*durations*)  
Change *durations* to nonreduced fractions with least common denominator:

```
>>> durations = [Duration(2, 4), 3, (5, 16)]
>>> for x in Duration.durations_to_nonreduced_fractions_with_common_denominator(
...     durations):
...     x
...
NonreducedFraction(8, 16)
NonreducedFraction(48, 16)
NonreducedFraction(5, 16)
```

Returns new object of *durations* type.

(Duration).**from\_lilypond\_duration\_string**(*lilypond\_duration\_string*)  
Initialize from LilyPond duration string:

```
>>> Duration.from_lilypond_duration_string('8.')
Duration(3, 16)
```

Returns duration.

(Duration).**is\_token**(*expr*)  
True if *expr* correctly initializes a duration. Otherwise false.

```
>>> Duration.is_token('8.')
True
```

Returns boolean.

(Duration).**yield\_durations** (*unique=False*)  
Yield durations.

**Example 1.** Yield all positive durations in Cantor diagonalized order:

```
>>> generator = Duration.yield_durations()
>>> for n in range(16):
...     generator.next()
...
Duration(1, 1)
Duration(2, 1)
Duration(1, 2)
Duration(1, 3)
Duration(1, 1)
Duration(3, 1)
Duration(4, 1)
Duration(3, 2)
Duration(2, 3)
Duration(1, 4)
Duration(1, 5)
Duration(1, 2)
Duration(1, 1)
Duration(2, 1)
Duration(5, 1)
Duration(6, 1)
```

**Example 2.** Yield all positive durations in Cantor diagonalized order uniquely:

```
>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     generator.next()
...
Duration(1, 1)
Duration(2, 1)
Duration(1, 2)
Duration(1, 3)
Duration(3, 1)
Duration(4, 1)
Duration(3, 2)
Duration(2, 3)
Duration(1, 4)
Duration(1, 5)
Duration(5, 1)
Duration(6, 1)
Duration(5, 2)
Duration(4, 3)
Duration(3, 4)
Duration(2, 5)
```

Returns generator.

## Special methods

```
(Duration).__abs__ (*args)
(Duration).__add__ (*args)
(Real).__complex__()
    complex(self) == complex(float(self), 0)
(Fraction).__copy__()
(Fraction).__deepcopy__ (memo)
(Duration).__div__ (*args)
```

(Duration) .\_\_divmod\_\_ (\*args)

(Duration) .\_\_eq\_\_ (arg)

(Rational) .\_\_float\_\_ ()

float(self) = self.numerator / self.denominator

It's important that this conversion use the integer's "true" division rather than casting one side to float before dividing so that ratios of huge integers convert without overflowing.

(Fraction) .\_\_floordiv\_\_ (a, b)

a // b

(Duration) .\_\_ge\_\_ (arg)

(Duration) .\_\_gt\_\_ (arg)

(Fraction) .\_\_hash\_\_ ()

hash(self)

Tricky because values that are exactly representable as a float must have the same hash as that float.

(Duration) .\_\_le\_\_ (arg)

(Duration) .\_\_lt\_\_ (arg)

(Duration) .\_\_mod\_\_ (\*args)

Multiplier.\_\_mul\_\_ (\*args)

Multiplier times duration gives duration.

Returns duration.

(Duration) .\_\_ne\_\_ (arg)

(Duration) .\_\_neg\_\_ (\*args)

(Duration) .\_\_new\_\_ (\*args)

(Fraction) .\_\_nonzero\_\_ (a)

a != 0

(Duration) .\_\_pos\_\_ (\*args)

(Duration) .\_\_pow\_\_ (\*args)

(Duration) .\_\_radd\_\_ (\*args)

(Duration) .\_\_rdiv\_\_ (\*args)

(Duration) .\_\_rdivmod\_\_ (\*args)

(Duration) .\_\_repr\_\_ ()

(Fraction) .\_\_rfloordiv\_\_ (b, a)

a // b

(Duration) .\_\_rmod\_\_ (\*args)

(Duration) .\_\_rmul\_\_ (\*args)

(Duration) .\_\_rpow\_\_ (\*args)

(Duration) .\_\_rsub\_\_ (\*args)

(Duration) .\_\_rtruediv\_\_ (\*args)

(Fraction) .\_\_str\_\_ ()

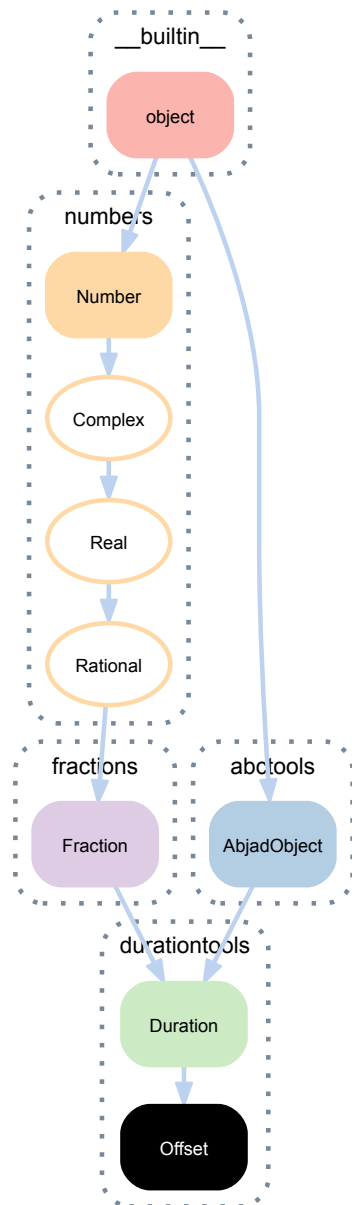
str(self)

(Duration) .\_\_sub\_\_ (\*args)

(Duration) .\_\_truediv\_\_ (\*args)

```
(Fraction).__trunc__(a)
trunc(a)
```

### 5.1.3 durationtools.Offset



**class** `durationtools.Offset`  
A musical offset.

```
>>> durationtools.Offset(121, 16)
Offset(121, 16)
```

Offset inherits from duration (which inherits from built-in Fraction).

#### Bases

- `durationtools.Duration`
- `abctools.AbjadObject`
- `fractions.Fraction`

- `numbers.Rational`
- `numbers.Real`
- `numbers.Complex`
- `numbers.Number`
- `__builtin__.object`

## Read-only properties

`(Fraction).denominator`

`(Duration).dot_count`

Positive integer number of dots required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         print '{}\t{}'.format(
...             duration.with_denominator(16), duration.dot_count)
...     except AssignabilityError:
...         print '{}\t{}'.format(
...             duration.with_denominator(16), '--')
...
1/16    0
2/16    0
3/16    1
4/16    0
5/16    --
6/16    1
7/16    2
8/16    0
9/16    --
10/16   --
11/16   --
12/16    1
13/16   --
14/16    2
15/16    3
16/16    0
```

Returns positive integer.

Raise assignability error when duration is not assignable.

`(Duration).equal_or_greater_assignable`

Equal or greater assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    3/8
6/16    3/8
7/16    7/16
8/16    1/2
9/16    3/4
10/16   3/4
11/16   3/4
12/16   3/4
13/16   7/8
14/16   7/8
```

```
15/16  15/16
16/16  1
```

Returns new duration.

(Duration).**equal\_or\_greater\_power\_of\_two**  
 Equal or greater power of 2:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    1/4
4/16    1/4
5/16    1/2
6/16    1/2
7/16    1/2
8/16    1/2
9/16    1
10/16   1
11/16   1
12/16   1
13/16   1
14/16   1
15/16   1
16/16   1
```

Returns new duration.

(Duration).**equal\_or\_lesser\_assignable**  
 Equal or lesser assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    1/4
6/16    3/8
7/16    7/16
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   3/4
13/16   3/4
14/16   7/8
15/16   15/16
16/16   1
```

Returns new duration.

(Duration).**equal\_or\_lesser\_power\_of\_two**  
 Equal or lesser power of 2:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_power_of_two
...     print '{}\t{}'.format(
...         duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
```

3/16	1/8
4/16	1/4
5/16	1/4
6/16	1/4
7/16	1/4
8/16	1/2
9/16	1/2
10/16	1/2
11/16	1/2
12/16	1/2
13/16	1/2
14/16	1/2
15/16	1/2
16/16	1

Returns new duration.

(Duration). **flag\_count**

Nonnegative integer number of flags required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     print '{}\t{}'.format(
...         duration.with_denominator(64), duration.flag_count)
...
1/64    4
2/64    3
3/64    3
4/64    2
5/64    2
6/64    2
7/64    2
8/64    1
9/64    1
10/64   1
11/64   1
12/64   1
13/64   1
14/64   1
15/64   1
16/64   0
```

Returns nonnegative integer.

(Duration). **has\_power\_of\_two\_denominator**

True when duration is an integer power of 2. Otherwise false:

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(1, n)
...     print '{}\t{}'.format(duration,
...         duration.has_power_of_two_denominator)
...
1      True
1/2    True
1/3    False
1/4    True
1/5    False
1/6    False
1/7    False
1/8    True
1/9    False
1/10   False
1/11   False
1/12   False
1/13   False
1/14   False
1/15   False
1/16   True
```

Returns boolean.

(Real). **imag**

Real numbers have no imaginary component.

(Duration) **.implied\_prolation**

Implied prolotion of multiplier:

```
>>> for denominator in range(1, 16 + 1):
...     multiplier = Multiplier(1, denominator)
...     print '{}\t{}'.format(multiplier,
...                             multiplier.implied_prolation)
...
1          1
1/2        1
1/3        2/3
1/4        1
1/5        4/5
1/6        2/3
1/7        4/7
1/8        1
1/9        8/9
1/10       4/5
1/11       8/11
1/12       2/3
1/13       8/13
1/14       4/7
1/15       8/15
1/16       1
```

Returns new multiplier.

(Duration) **.is\_assignable**

True when assignable. Otherwise false:

```
>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     print '{}\t{}'.format(duration.with_denominator(16),
...                             duration.is_assignable)
...
0/16      False
1/16      True
2/16      True
3/16      True
4/16      True
5/16      False
6/16      True
7/16      True
8/16      True
9/16      False
10/16     False
11/16     False
12/16     True
13/16     False
14/16     True
15/16     True
16/16     True
```

Returns boolean.

(Duration) **.lilypond\_duration\_string**

LilyPond duration string of assignable duration.

```
>>> Duration(3, 16).lilypond_duration_string
'8.'
```

Returns string.

Raise assignability error when duration is not assignable.

(Fraction) **.numerator**

(Duration) **.pair**

Pair of duration numerator and denominator:



```
>>> duration = Duration(3, 16)
```

```
>>> duration.pair
(3, 16)
```

Returns integer pair.

(Duration).**.prolation\_string**

Prolation string:

```
>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     rational = generator.next()
...     duration = Duration(rational)
...     print '{}\t{}'.format(duration, duration.prolation_string)
...
1      1:1
2      1:2
1/2    2:1
1/3    3:1
3      1:3
4      1:4
3/2    2:3
2/3    3:2
1/4    4:1
1/5    5:1
5      1:5
6      1:6
5/2    2:5
4/3    3:4
3/4    4:3
2/5    5:2
```

Returns string.

(Real).**.real**

Real numbers are their real component.

(Duration).**.reciprocal**

Reciprocal of duration.

Returns newly constructed duration.

(Duration).**.storage\_format**

Storage format of duration.

Returns string.

## Methods

(Real).**.conjugate()**

Conjugate is a no-op for Reals.

(Fraction).**.limit\_denominator** (*max\_denominator=1000000*)

Closest Fraction to self with denominator at most *max\_denominator*.

```
>>> Fraction('3.141592653589793').limit_denominator(10)
Fraction(22, 7)
>>> Fraction('3.141592653589793').limit_denominator(100)
Fraction(311, 99)
>>> Fraction(4321, 8765).limit_denominator(10000)
Fraction(4321, 8765)
```

(Duration).**.to\_clock\_string** (*escape\_ticks=False*)

Change duration to clock string.

**Example 1.** Change numeric *seconds* to clock string:

```
>>> duration = Duration(117)
>>> duration.to_clock_string()
'1\57''
```

**Example 2.** Change numeric *seconds* to escaped clock string:

```
>>> note = Note("c'4")
>>> clock_string = duration.to_clock_string(escape_ticks=True)
```

```
>>> markuptools.Markup('"%s"' % clock_string, Up)(note)
Markup(('1\57\'',), direction=Up)(c'4)
```

Returns string.

(Duration).**with\_denominator**(*denominator*)  
 Duration with *denominator*:

```
>>> duration = Duration(1, 4)

>>> for denominator in (4, 8, 16, 32):
...     print duration.with_denominator(denominator)
...
1/4
2/8
4/16
8/32
```

Returns new duration.

## Class methods

(Fraction).**from\_decimal**(*dec*)  
 Converts a finite Decimal instance to a rational number, exactly.

(Fraction).**from\_float**(*f*)  
 Converts a finite float to a rational number, exactly.  
 Beware that Fraction.from\_float(0.3) != Fraction(3, 10).

## Static methods

(Duration).**durations\_to\_nonreduced\_fractions\_with\_common\_denominator**(*durations*)  
 Change *durations* to nonreduced fractions with least common denominator:

```
>>> durations = [Duration(2, 4), 3, (5, 16)]
>>> for x in Duration.durations_to_nonreduced_fractions_with_common_denominator(
...     durations):
...     x
...
NonreducedFraction(8, 16)
NonreducedFraction(48, 16)
NonreducedFraction(5, 16)
```

Returns new object of *durations* type.

(Duration).**from\_lilypond\_duration\_string**(*lilypond\_duration\_string*)  
 Initialize from LilyPond duration string:

```
>>> Duration.from_lilypond_duration_string('8.')
Duration(3, 16)
```

Returns duration.

(Duration).**is\_token**(*expr*)  
 True if *expr* correctly initializes a duration. Otherwise false.

```
>>> Duration.is_token('8.')
True
```

Returns boolean.

(Duration).**yield\_durations** (*unique=False*)  
Yield durations.

**Example 1.** Yield all positive durations in Cantor diagonalized order:

```
>>> generator = Duration.yield_durations()
>>> for n in range(16):
...     generator.next()
...
Duration(1, 1)
Duration(2, 1)
Duration(1, 2)
Duration(1, 3)
Duration(1, 1)
Duration(3, 1)
Duration(4, 1)
Duration(3, 2)
Duration(2, 3)
Duration(1, 4)
Duration(1, 5)
Duration(1, 2)
Duration(1, 1)
Duration(2, 1)
Duration(5, 1)
Duration(6, 1)
```

**Example 2.** Yield all positive durations in Cantor diagonalized order uniquely:

```
>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     generator.next()
...
Duration(1, 1)
Duration(2, 1)
Duration(1, 2)
Duration(1, 3)
Duration(3, 1)
Duration(4, 1)
Duration(3, 2)
Duration(2, 3)
Duration(1, 4)
Duration(1, 5)
Duration(5, 1)
Duration(6, 1)
Duration(5, 2)
Duration(4, 3)
Duration(3, 4)
Duration(2, 5)
```

Returns generator.

## Special methods

```
(Duration).__abs__ (*args)
(Duration).__add__ (*args)
(Real).__complex__()
    complex(self) == complex(float(self), 0)
(Fraction).__copy__()
(Fraction).__deepcopy__ (memo)
(Duration).__div__ (*args)
```

```
(Duration) .__divmod__ (*args)
(Duration) .__eq__ (arg)
(Rational) .__float__ ()
    float(self) = self.numerator / self.denominator
```

It's important that this conversion use the integer's "true" division rather than casting one side to float before dividing so that ratios of huge integers convert without overflowing.

```
(Fraction) .__floordiv__ (a, b)
    a // b
(Duration) .__ge__ (arg)
(Duration) .__gt__ (arg)
(Fraction) .__hash__ ()
    hash(self)
```

Tricky because values that are exactly representable as a float must have the same hash as that float.

```
(Duration) .__le__ (arg)
(Duration) .__lt__ (arg)
(Duration) .__mod__ (*args)
(Duration) .__mul__ (*args)
(Duration) .__ne__ (arg)
(Duration) .__neg__ (*args)
(Duration) .__new__ (*args)
(Fraction) .__nonzero__ (a)
    a != 0
(Duration) .__pos__ (*args)
(Duration) .__pow__ (*args)
(Duration) .__radd__ (*args)
(Duration) .__rdiv__ (*args)
(Duration) .__rdivmod__ (*args)
(Duration) .__repr__ ()
(Fraction) .__rfloordiv__ (b, a)
    a // b
(Duration) .__rmod__ (*args)
(Duration) .__rmul__ (*args)
(Duration) .__rpow__ (*args)
(Duration) .__rsub__ (*args)
(Duration) .__rtruediv__ (*args)
(Fraction) .__str__ ()
    str(self)
```

```
Offset .__sub__ (expr)
    Offset taken from offset returns duration:
```

```
>>> durationtools.Offset(2) - durationtools.Offset(1, 2)
Duration(3, 2)
```

Duration taken from offset returns another offset:

```
>>> durationtools.Offset(2) - durationtools.Duration(1, 2)
Offset(3, 2)
```

Coerce *expr* to offset when *expr* is neither offset nor duration:

```
>>> durationtools.Offset(2) - Fraction(1, 2)
Duration(3, 2)
```

Returns duration or offset.

```
(Duration).__truediv__(*args)
```

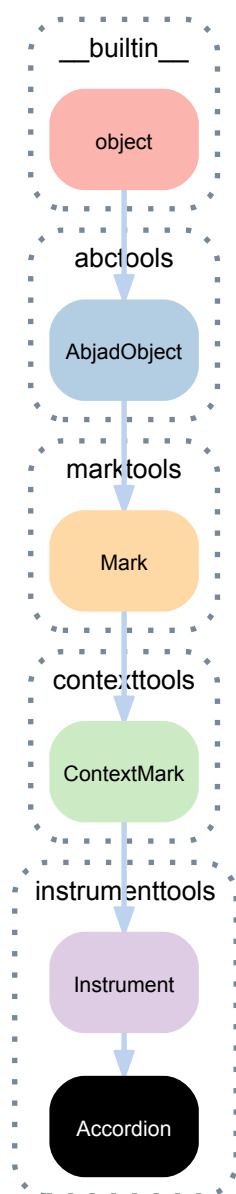
```
(Fraction).__trunc__(a)
trunc(a)
```



# INSTRUMENTTOOLS

## 6.1 Concrete classes

### 6.1.1 instrumenttools.Accordion



**class** `instrumenttools.Accordion` (*target\_context=None*, *\*\*kwargs*)  
 An accordion.

```
>>> piano_staff = scoretools.PianoStaff()
>>> piano_staff.append(Staff("c'8 d'8 e'8 f'8"))
>>> piano_staff.append(Staff("c'4 b4"))
>>> accordion = instrumenttools.Accordion()
>>> accordion = accordion.attach(piano_staff)
>>> show(piano_staff)
```



The accordion targets the piano staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
 Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
 LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
 Start component of mark.

Returns component or none.

Accordion **.storage\_format**  
 Accordion storage format.

Without customization:

```
>>> accordion = instrumenttools.Accordion()
>>> print accordion.storage_format
instrumenttools.Accordion()
```

With customization:

```
>>> custom = instrumenttools.Accordion()
>>> custom.instrument_name = 'fisarmonica'
>>> markup = markuptools.Markup('Fisarmonica')
>>> custom.instrument_name_markup = markup
>>> custom.short_instrument_name = 'fis.'
>>> markup = markuptools.Markup('Fis.')
>>> custom.short_instrument_name_markup = markup
>>> custom.allowable_clefs = ['treble']
>>> custom.pitch_range = '[C4, C6]'
>>> custom.sounding_pitch_of_written_middle_c = "c'"
```



```
>>> print custom.storage_format
instrumenttools.Accordion(
  instrument_name='fisarmonica',
  instrument_name_markup=markuptools.Markup((
    'Fisarmonica',
  )),
  short_instrument_name='fis.',
  short_instrument_name_markup=markuptools.Markup((
    'Fis.',
  )),
  allowable_clefs=contexttools.ClefMarkInventory([
    contexttools.ClefMark(
      'treble',
      target_context=stafftools.Staff
    )
  ]),
  pitch_range=pitchtools.PitchRange(
    '[C4, C6]'
  ),
  sounding_pitch_of_written_middle_c=pitchtools.NamedPitch("c'")
)
```

Returns string.

(ContextMark).**target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

Accordion.**allowable\_clefs**

Gets and sets allowable clefs.

Gets property:

```
>>> accordion.allowable_clefs
ClefMarkInventory([ClefMark('treble'), ClefMark('bass')])
```

```
>>> import copy
>>> skips = []
>>> for clef in accordion.allowable_clefs:
...     skip = skiptools.Skip((1, 8))
...     clef = copy.copy(clef)
...     clef = clef.attach(skip)
...     skips.append(skip)
>>> staff = Staff(skips)
>>> staff.override.clef.full_size_change = True
>>> staff.override.time_signature.stencil = False
>>> show(staff)
```



Sets property:

```
>>> accordion.allowable_clefs = ['treble']
>>> accordion.allowable_clefs
ClefMarkInventory([ClefMark('treble')])
```

Restores default:

```
>>> accordion.allowable_clefs = None
>>> accordion.allowable_clefs
ClefMarkInventory([ClefMark('treble'), ClefMark('bass')])
```

Returns clef inventory.

**Accordion.instrument\_name**

Gets and sets instrument name.

Gets property:

```
>>> accordion.instrument_name
'accordion'
```

Sets property:

```
>>> accordion.instrument_name = 'fisarmonica'
>>> accordion.instrument_name
'fisarmonica'
```

Restores default:

```
>>> accordion.instrument_name = None
>>> accordion.instrument_name
'accordion'
```

Returns string.

**Accordion.instrument\_name\_markup**

Gets and sets instrument name markup.

Gets property:

```
>>> accordion.instrument_name_markup
Markup(('Accordion',))
```

Sets property:

```
>>> markup = markuptools.Markup('Fisarmonica')
>>> accordion.instrument_name_markup = markup
>>> accordion.instrument_name_markup
Markup(('Fisarmonica',))
```

Restores default:

```
>>> accordion.instrument_name_markup = None
>>> accordion.instrument_name_markup
Markup(('Accordion',))
```

Returns markup.

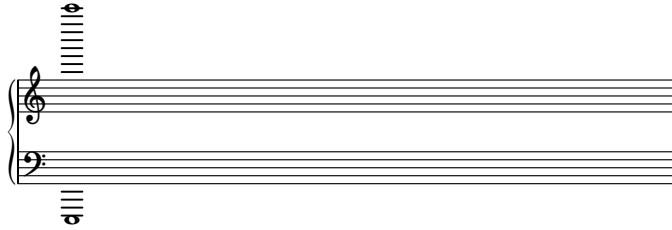
**Accordion.pitch\_range**

Gets and sets pitch range.

Gets property:

```
>>> accordion.pitch_range
PitchRange('E1, C8')
```

```
>>> result = scoretools.make_empty_piano_score()
>>> score, treble_staff, bass_staff = result
>>> note = Note("c'1")
>>> note.written_pitch = accordion.pitch_range.start_pitch
>>> bass_staff.append(note)
>>> note = Note("c'1")
>>> note.written_pitch = accordion.pitch_range.stop_pitch
>>> treble_staff.append(note)
>>> score.override.time_signature.stencil = False
>>> show(score)
```



Sets property:

```
>>> accordion.pitch_range = '[C2, C6]'
>>> accordion.pitch_range
PitchRange('[C2, C6]')
```

Restores default:

```
>>> accordion.pitch_range = None
>>> accordion.pitch_range
PitchRange('[E1, C8]')
```

Returns pitch range.

**Accordion.short\_instrument\_name**

Gets and sets short instrument name.

Gets property:

```
>>> accordion.short_instrument_name
'acc.'
```

Sets property:

```
>>> accordion.short_instrument_name = 'fis.'
>>> accordion.short_instrument_name
'fis.'
```

Restores default:

```
>>> accordion.short_instrument_name = None
>>> accordion.short_instrument_name
'acc.'
```

Returns string.

**Accordion.short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Gets property:

```
>>> accordion.short_instrument_name_markup
Markup(('Acc.',))
```

Sets property:

```
>>> markup = markuptools.Markup('fis.')
>>> accordion.short_instrument_name_markup = markup
>>> accordion.short_instrument_name_markup
Markup(('fis.',))
```

Restores default:

```
>>> accordion.short_instrument_name_markup = None
>>> accordion.short_instrument_name_markup
Markup(('Acc.',))
```

Returns markup.

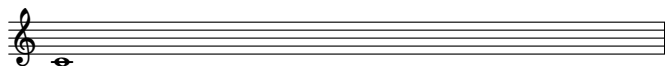
**Accordion.sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Gets property:

```
>>> accordion.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> pitch = accordion.sounding_pitch_of_written_middle_c
>>> note = Note(pitch, Duration(1))
>>> staff = Staff([note])
>>> staff.override.time_signature.stencil = False
>>> show(staff)
```



Sets property:

```
>>> accordion.sounding_pitch_of_written_middle_c = 'cs'
>>> accordion.sounding_pitch_of_written_middle_c
NamedPitch('cs')
```

Restores default:

```
>>> accordion.sounding_pitch_of_written_middle_c = None
>>> accordion.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

Returns named pitch.

## Methods

(ContextMark) **.attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) **.\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) **.\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) **.\_\_hash\_\_**()

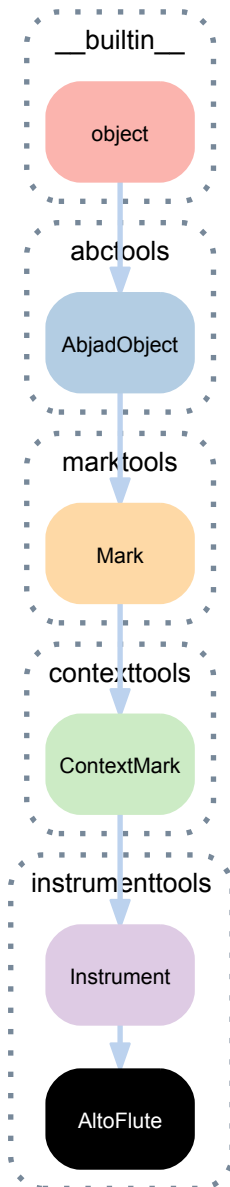
Hash value of instrument.

Returns integer.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(Instrument).__repr__()`  
 Interpreter representation of instrument.  
 Returns string.

### 6.1.2 instrumenttools.AltoFlute

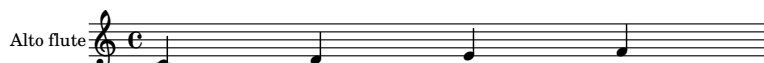


**class** `instrumenttools.AltoFlute` (*\*\*kwargs*)  
 An alto flute.

```

>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> alto_flute = instrumenttools.AltoFlute()
>>> alto_flute.attach(staff)
>>> show(staff)

```



The alto flute targets the staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark).**effective\_context**

Effective context of context mark.

Returns context mark or none.

(Instrument).**lilypond\_format**

LilyPond format of instrument mark.

Returns string.

(Mark).**start\_component**

Start component of mark.

Returns component or none.

AltoFlute.**storage\_format**

Alto flute storage format.

Without customization:

```
>>> alto_flute = instrumenttools.AltoFlute()
>>> print alto_flute.storage_format
instrumenttools.AltoFlute()
```

With customization:

```
>>> custom = instrumenttools.AltoFlute()
>>> custom.instrument_name = 'flauto contralto'
>>> markup = markuptools.Markup('Flauto contralto')
>>> custom.instrument_name_markup = markup
>>> custom.short_instrument_name = 'fl. contr.'
>>> markup = markuptools.Markup('Fl. contr.')
>>> custom.short_instrument_name_markup = markup
>>> custom.pitch_range = '[G3, C7]'
```

```
>>> print custom.storage_format
instrumenttools.AltoFlute(
  instrument_name='flauto contralto',
  instrument_name_markup=markuptools.Markup((
    'Flauto contralto',
  )),
  short_instrument_name='fl. contr.',
  short_instrument_name_markup=markuptools.Markup((
    'Fl. contr.',
  )),
  pitch_range=pitchtools.PitchRange(
    '[G3, C7]'
  )
)
```

Returns string.

(ContextMark).**target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

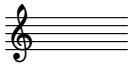
AltoFlute.**allowable\_clefs**

Gets and sets allowable clefs.

Gets property:

```
>>> alto_flute.allowable_clefs
ClefMarkInventory([ClefMark('treble')])
```

```
>>> import copy
>>> skips = []
>>> for clef in alto_flute.allowable_clefs:
...     skip = skiptools.Skip((1, 8))
...     clef = copy.copy(clef)
...     clef = clef.attach(skip)
...     skips.append(skip)
>>> staff = Staff(skips)
>>> staff.override.clef.full_size_change = True
>>> staff.override.time_signature.stencil = False
>>> show(staff)
```



Sets property:

```
>>> alto_flute.allowable_clefs = ['treble', 'treble^8']
>>> alto_flute.allowable_clefs
ClefMarkInventory([ClefMark('treble'), ClefMark('treble^8')])
```

Restores default:

```
>>> alto_flute.allowable_clefs = None
>>> alto_flute.allowable_clefs
ClefMarkInventory([ClefMark('treble')])
```

Returns clef inventory.

AltoFlute.**instrument\_name**

Gets and sets instrument name.

Gets property:

```
>>> alto_flute.instrument_name
'alto flute'
```

Sets property:

```
>>> alto_flute.instrument_name = 'flauto contralto'
>>> alto_flute.instrument_name
'flauto contralto'
```

Restores default:

```
>>> alto_flute.instrument_name = None
>>> alto_flute.instrument_name
'alto flute'
```

Returns string.

AltoFlute.**instrument\_name\_markup**

Gets and sets instrument name markup.

Gets property:

```
>>> alto_flute.instrument_name_markup
Markup(('Alto flute',))
```

Sets property:

```
>>> markup = markuptools.Markup('Flauto contralto')
>>> alto_flute.instrument_name_markup = markup
>>> alto_flute.instrument_name_markup
Markup(('Flauto contralto',))
```

Restores default:

```
>>> alto_flute.instrument_name_markup = None
>>> alto_flute.instrument_name_markup
Markup(('Alto flute',))
```

Returns markup.

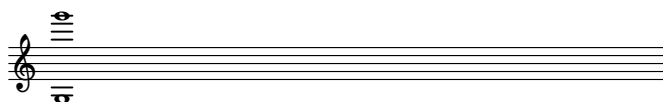
**AltoFlute.pitch\_range**

Gets and sets pitch range.

Gets property:

```
>>> alto_flute.pitch_range
PitchRange(' [G3, G6]')
```

```
>>> chord = Chord("<c' d'>1")
>>> start_pitch = alto_flute.pitch_range.start_pitch
>>> chord[0].written_pitch = start_pitch
>>> stop_pitch = alto_flute.pitch_range.stop_pitch
>>> chord[1].written_pitch = stop_pitch
>>> voice = Voice([chord])
>>> staff = Staff([voice])
>>> staff.override.time_signature.stencil = False
>>> show(staff)
```



Sets property:

```
>>> alto_flute.pitch_range = ' [G3, C7]\'
>>> alto_flute.pitch_range
PitchRange(' [G3, C7]')
```

Restores default:

```
>>> alto_flute.pitch_range = None
>>> alto_flute.pitch_range
PitchRange(' [G3, G6]')
```

Returns pitch range.

**AltoFlute.short\_instrument\_name**

Gets and sets short instrument name.

Gets property:

```
>>> alto_flute.short_instrument_name
'alt. fl.'
```

Sets property:

```
>>> alto_flute.short_instrument_name = 'fl. contr.'
>>> alto_flute.short_instrument_name
'fl. contr.'
```

Restores default:



```
>>> alto_flute.short_instrument_name = None
>>> alto_flute.short_instrument_name
'alt. fl.'
```

Returns string.

**AltoFlute.short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Gets property:

```
>>> alto_flute.short_instrument_name_markup
Markup(('Alt. fl.',))
```

Sets property:

```
>>> markup = markuptools.Markup('Fl. contr.')
>>> alto_flute.short_instrument_name_markup = markup
>>> alto_flute.short_instrument_name_markup
Markup(('Fl. contr.',))
```

Restores default:

```
>>> alto_flute.short_instrument_name_markup = None
>>> alto_flute.short_instrument_name_markup
Markup(('Alt. fl.',))
```

Returns markup.

**AltoFlute.sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Gets property:

```
>>> alto_flute.sounding_pitch_of_written_middle_c
NamedPitch('g')
```

```
>>> pitch = alto_flute.sounding_pitch_of_written_middle_c
>>> note = Note(pitch, Duration(1))
>>> voice = Voice([note])
>>> staff = Staff([voice])
>>> staff.override.time_signature.stencil = False
>>> show(staff)
```



Sets property:

```
>>> alto_flute.sounding_pitch_of_written_middle_c = 'gs'
>>> alto_flute.sounding_pitch_of_written_middle_c
NamedPitch('gs')
```

Restores default:

```
>>> alto_flute.sounding_pitch_of_written_middle_c = None
>>> alto_flute.sounding_pitch_of_written_middle_c
NamedPitch('g')
```

Returns named pitch.

## Methods

(ContextMark) **.attach**(start\_component)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

`(ContextMark).detach()`

Detaches context mark.

Returns context mark.

## Special methods

`(Mark).__call__(*args)`

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

True when instrument equals *arg*. Otherwise false.

Returns boolean.

`(Instrument).__hash__()`

Hash value of instrument.

Returns integer.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

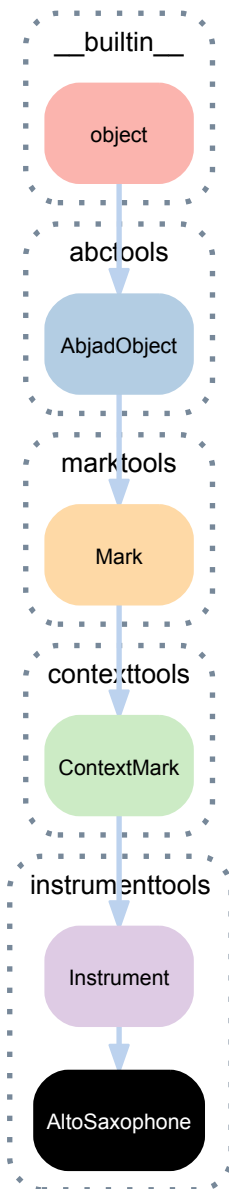
Returns boolean.

`(Instrument).__repr__()`

Interpreter representation of instrument.

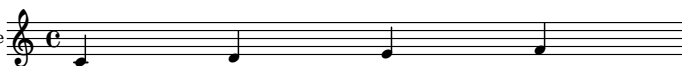
Returns string.

### 6.1.3 instrumenttools.AltoSaxophone



**class** instrumenttools.**AltoSaxophone** (\*\*kwargs)  
 An alto saxophone.

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> alto_sax = instrumenttools.AltoSaxophone()
>>> alto_sax = alto_sax.attach(staff)
>>> show(staff)
```

Alto saxophone 

The alto saxophone targets staff context by default.

#### Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark).**effective\_context**

Effective context of context mark.

Returns context mark or none.

(Instrument).**lilypond\_format**

LilyPond format of instrument mark.

Returns string.

(Mark).**start\_component**

Start component of mark.

Returns component or none.

AltoSaxophone.**storage\_format**

Alto sax storage format.

Without customization:

```
>>> alto_sax = instrumenttools.AltoSaxophone()
>>> print alto_sax.storage_format
instrumenttools.AltoSaxophone()
```

With customization:

```
>>> custom = instrumenttools.AltoSaxophone()
>>> custom.instrument_name = 'sassofono contralto'
>>> markup = markuptools.Markup('Sassofono contralto')
>>> custom.instrument_name_markup = markup
>>> custom.short_instrument_name = 'sass. contr.'
>>> markup = markuptools.Markup('Sass. contr.')
>>> custom.short_instrument_name_markup = markup
>>> custom.pitch_range = '[G3, C7]'
```

```
>>> print custom.storage_format
instrumenttools.AltoSaxophone(
    instrument_name='sassofono contralto',
    instrument_name_markup=markuptools.Markup((
        'Sassofono contralto',
    )),
    short_instrument_name='sass. contr.',
    short_instrument_name_markup=markuptools.Markup((
        'Sass. contr.',
    )),
    pitch_range=pitchtools.PitchRange(
        '[G3, C7]'
    )
)
```

Returns string.

(ContextMark).**target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

AltoSaxophone.**allowable\_clefs**

Gets and sets allowable clefs.

Gets property:

```
>>> alto_sax.allowable_clefs
ClefMarkInventory([ClefMark('treble')])
```

```
>>> import copy
>>> skips = []
>>> for clef in alto_sax.allowable_clefs:
...     skip = skiptools.Skip((1, 8))
...     clef = copy.copy(clef)
...     clef = clef.attach(skip)
...     skips.append(skip)
>>> staff = Staff(skips)
>>> staff.override.clef.full_size_change = True
>>> staff.override.time_signature.stencil = False
>>> show(staff)
```



Sets property:

```
>>> alto_sax.allowable_clefs = ['treble', 'treble^8']
>>> alto_sax.allowable_clefs
ClefMarkInventory([ClefMark('treble'), ClefMark('treble^8')])
```

Restores default:

```
>>> alto_sax.allowable_clefs = None
>>> alto_sax.allowable_clefs
ClefMarkInventory([ClefMark('treble')])
```

Returns clef inventory.

`AltoSaxophone.instrument_name`

Gets and sets instrument name.

Gets property:

```
>>> alto_sax.instrument_name
'alto saxophone'
```

Sets property:

```
>>> alto_sax.instrument_name = 'sassofono contralto'
>>> alto_sax.instrument_name
'sassofono contralto'
```

Restores default:

```
>>> alto_sax.instrument_name = None
>>> alto_sax.instrument_name
'alto saxophone'
```

Returns string.

`AltoSaxophone.instrument_name_markup`

Gets and sets instrument name markup.

Gets property:

```
>>> alto_sax.instrument_name_markup
Markup(('Alto saxophone',))
```

Sets property:

```
>>> markup = markuptools.Markup('Sassofono contralto')
>>> alto_sax.instrument_name_markup = markup
>>> alto_sax.instrument_name_markup
Markup(('Sassofono contralto',))
```

Restores default:

```
>>> alto_sax.instrument_name_markup = None
>>> alto_sax.instrument_name_markup
Markup(('Alto saxophone',))
```

Returns markup.

**AltoSaxophone.pitch\_range**

Gets and sets pitch range.

Gets property:

```
>>> alto_sax.pitch_range
PitchRange(' [C#3, A5]')
```

```
>>> chord = Chord("<c' d'>1")
>>> start_pitch = alto_sax.pitch_range.start_pitch
>>> chord[0].written_pitch = start_pitch
>>> stop_pitch = alto_sax.pitch_range.stop_pitch
>>> chord[1].written_pitch = stop_pitch
>>> voice = Voice([chord])
>>> staff = Staff([voice])
>>> staff.override.time_signature.stencil = False
>>> show(staff)
```



Sets property:

```
>>> alto_sax.pitch_range = ' [C#3, A6] '
>>> alto_sax.pitch_range
PitchRange(' [C#3, A6]')
```

Restores default:

```
>>> alto_sax.pitch_range = None
>>> alto_sax.pitch_range
PitchRange(' [C#3, A5]')
```

Returns pitch range.

**AltoSaxophone.short\_instrument\_name**

Gets and sets short instrument name.

Gets property:

```
>>> alto_sax.short_instrument_name
'alt. sax.'
```

Sets property:

```
>>> alto_sax.short_instrument_name = 'sass. contr.'
>>> alto_sax.short_instrument_name
'sass. contr.'
```

Restores default:

```
>>> alto_sax.short_instrument_name = None
>>> alto_sax.short_instrument_name
'alt. sax.'
```

Returns string.

**AltoSaxophone.short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Gets property:

```
>>> alto_sax.short_instrument_name_markup
Markup(('Alt. sax.',))
```

Sets property:

```
>>> markup = markuptools.Markup('Sass. contr.')
>>> alto_sax.short_instrument_name_markup = markup
>>> alto_sax.short_instrument_name_markup
Markup(('Sass. contr.',))
```

Restores default:

```
>>> alto_sax.short_instrument_name_markup = None
>>> alto_sax.short_instrument_name_markup
Markup(('Alt. sax.',))
```

Returns markup.

`AltoSaxophone.sounding_pitch_of_written_middle_c`

Gets and sets sounding pitch of written middle C.

Gets property:

```
>>> alto_sax.sounding_pitch_of_written_middle_c
NamedPitch('ef')
```

Sets property:

```
>>> alto_sax.sounding_pitch_of_written_middle_c = 'e'
>>> alto_sax.sounding_pitch_of_written_middle_c
NamedPitch('e')
```

Restores default:

```
>>> alto_sax.sounding_pitch_of_written_middle_c = None
>>> alto_sax.sounding_pitch_of_written_middle_c
NamedPitch('ef')
```

Returns named pitch.

## Methods

`(ContextMark).attach(start_component)`

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

`(ContextMark).detach()`

Detaches context mark.

Returns context mark.

## Special methods

`(Mark).__call__(*args)`

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

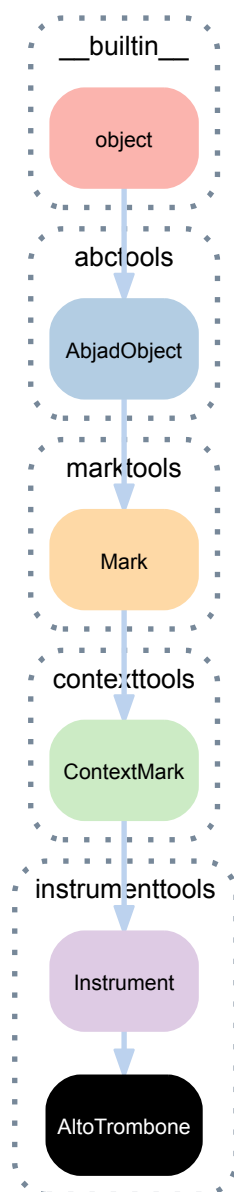
`(Instrument).__eq__(arg)`  
 True when instrument equals *arg*. Otherwise false.  
 Returns boolean.

`(Instrument).__hash__()`  
 Hash value of instrument.  
 Returns integer.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(Instrument).__repr__()`  
 Interpreter representation of instrument.  
 Returns string.

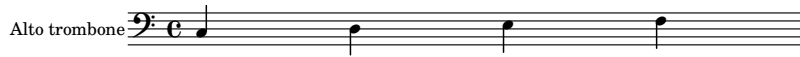
### 6.1.4 instrumenttools.AltoTrombone





**class** instrumenttools.**AltoTrombone** (*\*\*kwargs*)  
 An alto trombone.

```
>>> staff = Staff("c4 d4 e4 f4")
>>> clef = contexttools.ClefMark('bass')
>>> clef = clef.attach(staff)
>>> alto_trombone = instrumenttools.AltoTrombone()
>>> alto_trombone = alto_trombone.attach(staff)
>>> show(staff)
```



The alto trombone targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
 Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
 LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
 Start component of mark.

Returns component or none.

AltoTrombone **.storage\_format**  
 Storage format.

Without customization:

```
>>> alto_trombone = instrumenttools.AltoTrombone()
>>> print alto_trombone.storage_format
instrumenttools.AltoTrombone()
```

With customization:

```
>>> custom = instrumenttools.AltoTrombone()
>>> custom.instrument_name = 'trombone contralto'
>>> markup = markuptools.Markup('Trombone contralto')
>>> custom.instrument_name_markup = markup
>>> custom.short_instrument_name = 'trb. contr.'
>>> markup = markuptools.Markup('Trb. contr.')
>>> custom.short_instrument_name_markup = markup
>>> custom.pitch_range = '[A2, C6]'
>>> custom.sounding_pitch_of_written_middle_c = 'ef'
```

```
>>> print custom.storage_format
instrumenttools.AltoTrombone(
    instrument_name='trombone contralto',
    instrument_name_markup=markuptools.Markup((
        'Trombone contralto',
    )),
    short_instrument_name='trb. contr.',
    short_instrument_name_markup=markuptools.Markup((
        'Trb. contr.',
    )),
    pitch_range=pitchtools.PitchRange(
        '[A2, C6]'
    ),
    sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('ef')
)
```

Returns string.

(ContextMark).**target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

AltoTrombone.**allowable\_clefs**

Gets and sets allowable clefs.

Gets property:

```
>>> alto_trombone.allowable_clefs
ClefMarkInventory([ClefMark('bass'), ClefMark('tenor')])
```

```
>>> import copy
>>> skips = []
>>> for clef in alto_trombone.allowable_clefs:
...     skip = skiptools.Skip((1, 8))
...     clef = copy.copy(clef)
...     clef = clef.attach(skip)
...     skips.append(skip)
>>> staff = Staff(skips)
>>> staff.override.clef.full_size_change = True
>>> staff.override.time_signature.stencil = False
>>> show(staff)
```



Sets property:

```
>>> alto_trombone.allowable_clefs = ['treble']
>>> alto_trombone.allowable_clefs
ClefMarkInventory([ClefMark('treble')])
```

Restores default:

```
>>> alto_trombone.allowable_clefs = None
>>> alto_trombone.allowable_clefs
ClefMarkInventory([ClefMark('bass'), ClefMark('tenor')])
```

Returns clef inventory.

AltoTrombone.**instrument\_name**

Gets and sets instrument name.

Gets property:

```
>>> alto_trombone.instrument_name
'alto trombone'
```

Sets property:

```
>>> alto_trombone.instrument_name = 'trombone contralto'
>>> alto_trombone.instrument_name
'trombone contralto'
```

Restores default:

```
>>> alto_trombone.instrument_name = None
>>> alto_trombone.instrument_name
'alto trombone'
```

Returns string.

`AltoTrombone.instrument_name_markup`

Gets and sets instrument name markup.

Gets property:

```
>>> alto_trombone.instrument_name_markup
Markup(('Alto trombone',))
```

Sets property:

```
>>> markup = markuptools.Markup('Trombone contralto')
>>> alto_trombone.instrument_name_markup = markup
>>> alto_trombone.instrument_name_markup
Markup(('Trombone contralto',))
```

Restores default:

```
>>> alto_trombone.instrument_name_markup = None
>>> alto_trombone.instrument_name_markup
Markup(('Alto trombone',))
```

Returns markup.

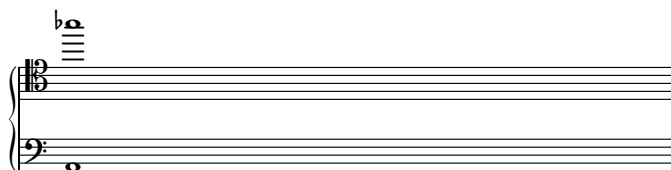
`AltoTrombone.pitch_range`

Gets and sets pitch range.

Gets property:

```
>>> alto_trombone.pitch_range
PitchRange(' [A2, Bb5]')
```

```
>>> result = scoretools.make_empty_piano_score()
>>> score, tenor_staff, bass_staff = result
>>> clef = inspect(tenor_staff).get_effective_context_mark(
...     contexttools.ClefMark)
>>> clef.clef_name = 'tenor'
>>> note = Note("c'1")
>>> start_pitch = alto_trombone.pitch_range.start_pitch
>>> note.written_pitch = start_pitch
>>> voice = Voice([note])
>>> bass_staff.append(voice)
>>> note = Note("c'1")
>>> stop_pitch = alto_trombone.pitch_range.stop_pitch
>>> note.written_pitch = stop_pitch
>>> voice = Voice([note])
>>> tenor_staff.append(voice)
>>> score.override.time_signature.stencil = False
>>> show(score)
```



Sets property:

```
>>> alto_trombone.pitch_range = '[A2, C6]'
>>> alto_trombone.pitch_range
PitchRange('[A2, C6]')
```

Restores default:

```
>>> alto_trombone.pitch_range = None
>>> alto_trombone.pitch_range
PitchRange('[A2, Bb5]')
```

Returns pitch range.

**AltoTrombone.short\_instrument\_name**

Gets and sets short instrument name.

Gets property:

```
>>> alto_trombone.short_instrument_name
'alt. trb.'
```

Sets property:

```
>>> alto_trombone.short_instrument_name = 'trb. contr.'
>>> alto_trombone.short_instrument_name
'trb. contr.'
```

Restores default:

```
>>> alto_trombone.short_instrument_name = None
>>> alto_trombone.short_instrument_name
'alt. trb.'
```

Returns string.

**AltoTrombone.short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Gets property:

```
>>> alto_trombone.short_instrument_name_markup
Markup(('Alt. trb.',))
```

Sets property:

```
>>> markup = markuptools.Markup('Trb. contr.')
>>> alto_trombone.short_instrument_name_markup = markup
>>> alto_trombone.short_instrument_name_markup
Markup(('Trb. contr.',))
```

Restores default:

```
>>> alto_trombone.short_instrument_name_markup = None
>>> alto_trombone.short_instrument_name_markup
Markup(('Alt. trb.',))
```

Returns markup.

**AltoTrombone.sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Gets property:

```
>>> alto_trombone.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

Sets property:

```
>>> alto_trombone.sounding_pitch_of_written_middle_c = 'ef'
>>> alto_trombone.sounding_pitch_of_written_middle_c
NamedPitch('ef')
```

Restores default:

```
>>> alto_trombone.sounding_pitch_of_written_middle_c = None
>>> alto_trombone.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

Returns named pitch.

## Methods

(ContextMark) **.attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start* component.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) **.\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) **.\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) **.\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) **.\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

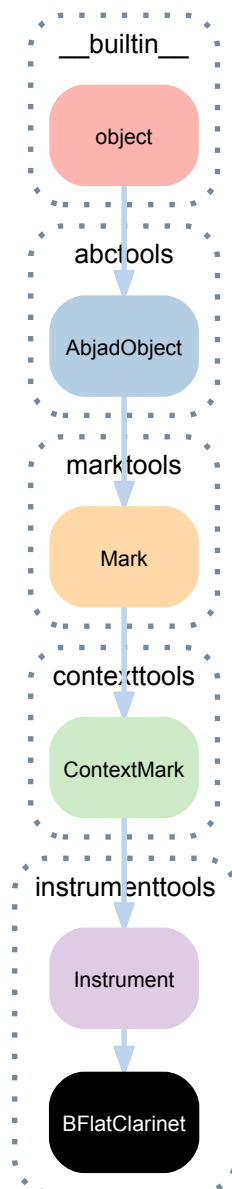
Returns boolean.

(Instrument) **.\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.5 instrumenttools.BFlatClarinet



**class** instrumenttools.**BFlatClarinet** (\*\*kwargs)  
A B-flat clarinet.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> clarinet = instrumenttools.BFlatClarinet() (staff)
>>> clarinet
BFlatClarinet() (Staff{4})
```

```
>>> show(staff)
```

Clarinet in B-flat 

The B-flat clarinet targets staff context by default.

#### Bases

- instrumenttools.Instrument

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

(Instrument) .**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach** (*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach** ()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_** (\**args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_** (\**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_** (*arg*)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_** ()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

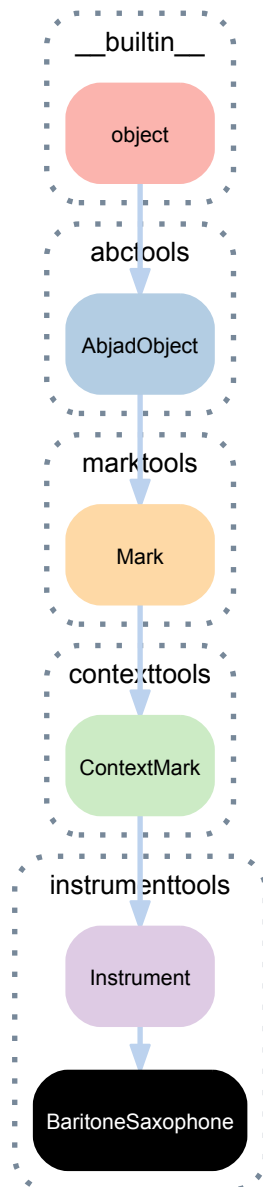
(Instrument) .**\_\_repr\_\_** ()

Interpreter representation of instrument.

Returns string.



### 6.1.6 instrumenttools.BaritoneSaxophone



**class** instrumenttools.**BaritoneSaxophone** (*\*\*kwargs*)  
 A baritone saxophone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> baritone_sax = instrumenttools.BaritoneSaxophone()
>>> baritone_sax.attach(staff)
>>> show(staff)
```



The baritone saxophone targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument).**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

BaritoneSaxophone.**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

```
>>> baritone_sax.sounding_pitch_of_written_middle_c
NamedPitch('ef,')
```

```
>>> baritone_sax.sounding_pitch_of_written_middle_c = 'e'
>>> baritone_sax.sounding_pitch_of_written_middle_c
NamedPitch('e')
```

```
>>> baritone_sax.sounding_pitch_of_written_middle_c = None
>>> baritone_sax.sounding_pitch_of_written_middle_c
NamedPitch('ef,')
```

Returns named pitch.

## Methods

(ContextMark).**attach**(start\_component)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

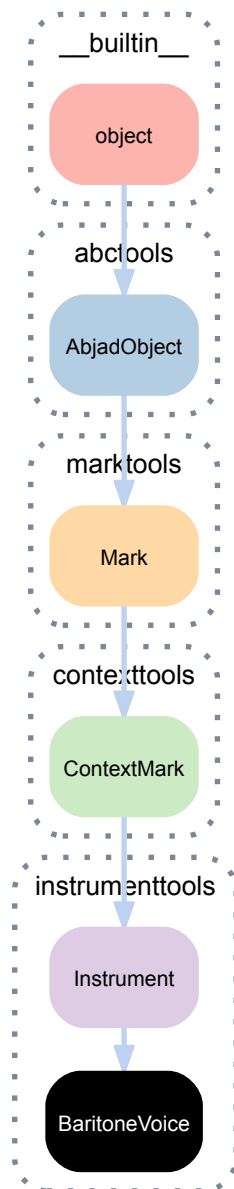
(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(Instrument).__repr__()`  
 Interpreter representation of instrument.  
 Returns string.

## 6.1.7 instrumenttools.BaritoneVoice



**class instrumenttools.BaritoneVoice** *(\*\*kwargs)*  
 A baritone voice.

```
>>> staff = Staff("c8 d8 e8 f8")
>>> show(staff)
```



```
>>> baritone_voice = instrumenttools.BaritoneVoice()
>>> baritone_voice = baritone_voice.attach(staff)
>>> show(staff)
```



The baritone voice targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument).**short\_instrument\_name**

Gets and sets short instrument name.

Returns string.

(Instrument).**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

BaritoneVoice.**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

```
>>> baritone_voice.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> baritone_voice.sounding_pitch_of_written_middle_c = 'g'
>>> baritone_voice.sounding_pitch_of_written_middle_c
NamedPitch('g')
```

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

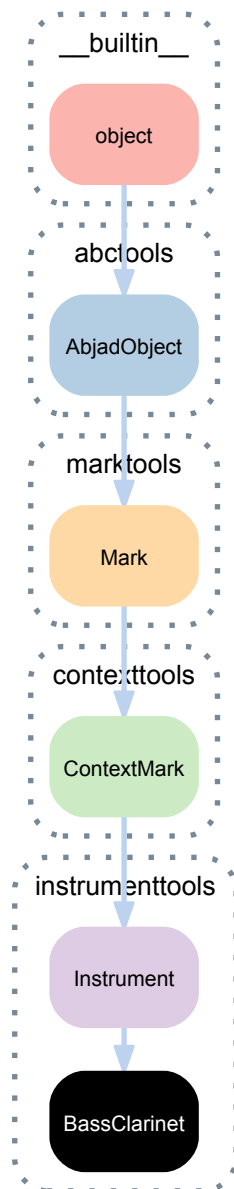
(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(Instrument).__repr__()`  
 Interpreter representation of instrument.  
 Returns string.

### 6.1.8 instrumenttools.BassClarinet



**class instrumenttools.BassClarinet** (*\*\*kwargs*)  
 A bass clarinet.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> bass_clarinet = instrumenttools.BassClarinet()
>>> bass_clarinet = bass_clarinet.attach(staff)
>>> show(staff)
```



The bass clarinet targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.



(Instrument).**short\_instrument\_name**

Gets and sets short instrument name.

Returns string.

(Instrument).**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

BassClarinet.**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

```
>>> bass_clarinet.sounding_pitch_of_written_middle_c
NamedPitch('bf,')
```

```
>>> bass_clarinet.sounding_pitch_of_written_middle_c = 'b,'
>>> bass_clarinet.sounding_pitch_of_written_middle_c
NamedPitch('b,')
```

```
>>> bass_clarinet.sounding_pitch_of_written_middle_c = None
>>> bass_clarinet.sounding_pitch_of_written_middle_c
NamedPitch('bf,')
```

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

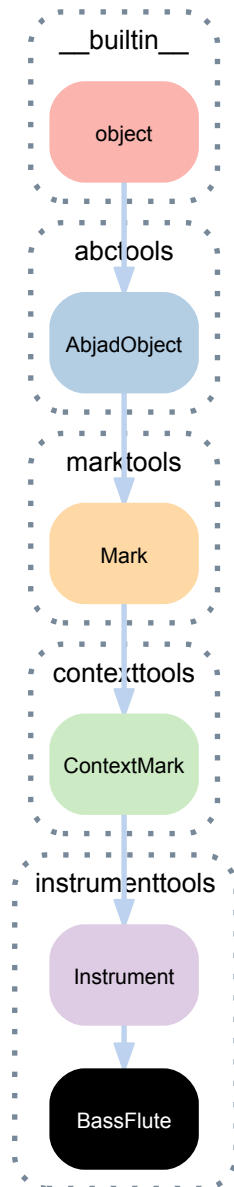
Hash value of instrument.

Returns integer.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(Instrument).__repr__()`  
 Interpreter representation of instrument.  
 Returns string.

### 6.1.9 instrumenttools.BassFlute



**class** `instrumenttools.BassFlute` (*\*\*kwargs*)  
 A bass flute.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> bass_flute = instrumenttools.BassFlute()
>>> bass_flute = bass_flute.attach(staff)
>>> show(staff)
```



The bass flute targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**

Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**

Gets and sets short instrument name.

Returns string.

(Instrument) **.short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

BassFlute **.sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

```
>>> bass_flute.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> bass_flute.sounding_pitch_of_written_middle_c = 'cs'
>>> bass_flute.sounding_pitch_of_written_middle_c
NamedPitch('cs')
```

```
>>> bass_flute.sounding_pitch_of_written_middle_c = None
>>> bass_flute.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

Returns named pitch.

## Methods

(ContextMark) **.attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) **.\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) **.\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

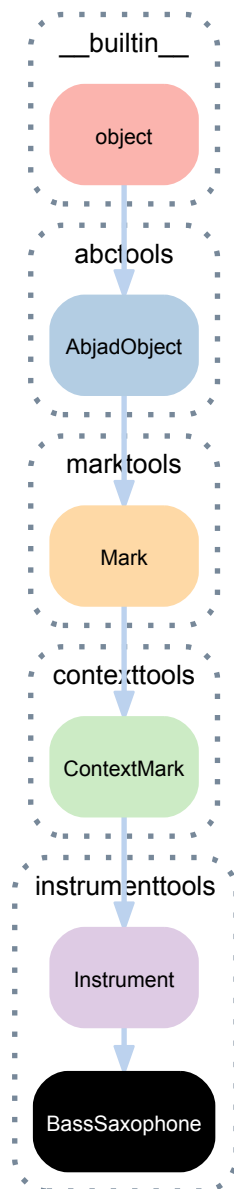
Returns boolean.

(Instrument).**\_\_hash\_\_**()  
 Hash value of instrument.  
 Returns integer.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Instrument).**\_\_repr\_\_**()  
 Interpreter representation of instrument.  
 Returns string.

### 6.1.10 instrumenttools.BassSaxophone



**class** instrumenttools.**BassSaxophone**(\*\**kwargs*)  
 A bass saxophone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> bass_sax = instrumenttools.BassSaxophone()  
>>> bass_sax = bass_sax.attach(staff)  
>>> show(staff)
```



The bass saxophone targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`

Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`

Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`

Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`

Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

`(ContextMark).attach(start_component)`

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

`(ContextMark).detach()`

Detaches context mark.

Returns context mark.

## Special methods

`(Mark).__call__(*args)`

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

True when instrument equals *arg*. Otherwise false.

Returns boolean.

`(Instrument).__hash__()`

Hash value of instrument.

Returns integer.

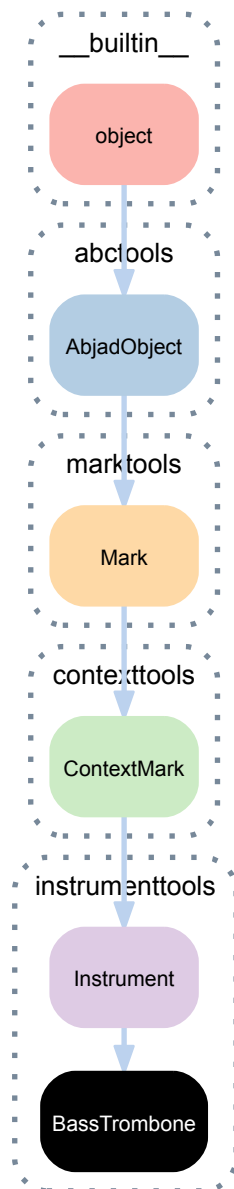
`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(Instrument).__repr__()`  
 Interpreter representation of instrument.  
 Returns string.

### 6.1.11 instrumenttools.BassTrombone



**class instrumenttools.BassTrombone** (*\*\*kwargs*)  
 A bass trombone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.BassTrombone()(staff)
BassTrombone()(Staff{4})
```

```
>>> show(staff)
```

Bass trombone



The tenor trombone targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument) .**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

(Instrument) .**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

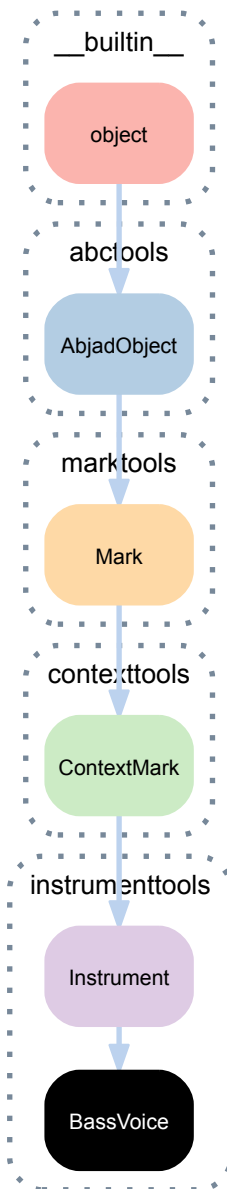
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.12 instrumenttools.BassVoice



**class** instrumenttools.**BassVoice** (*\*\*kwargs*)  
 A bass.

```
>>> staff = Staff("c8 d8 e8 f8")
```

```
>>> instrumenttools.BassVoice() (staff)
BassVoice() (Staff{4})
```

```
>>> show(staff)
```



The bass voice targets staff context by default.

#### Bases

- `instrumenttools.Instrument`

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

### Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

(Instrument).**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

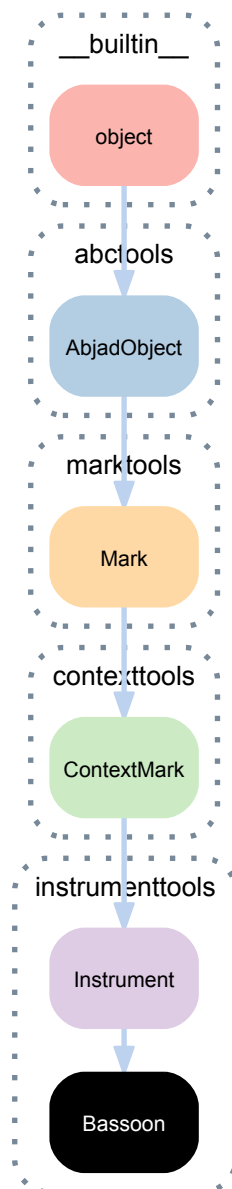
Returns boolean.

(Instrument).**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.13 instrumenttools.Bassoon



**class** instrumenttools.**Bassoon** (\*\*kwargs)  
A bassoon.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> clef = contexttools.ClefMark('bass')
>>> clef = clef.attach(staff)
>>> show(staff)
```



```
>>> bassoon = instrumenttools.Bassoon()
>>> bassoon = bassoon.attach(staff)
```

The bassoon targets staff context by default.

#### Bases

- `instrumenttools.Instrument`

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

(Instrument) .**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach** (*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach** ()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_** (\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_** (\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_** (arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_** ()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_** (expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

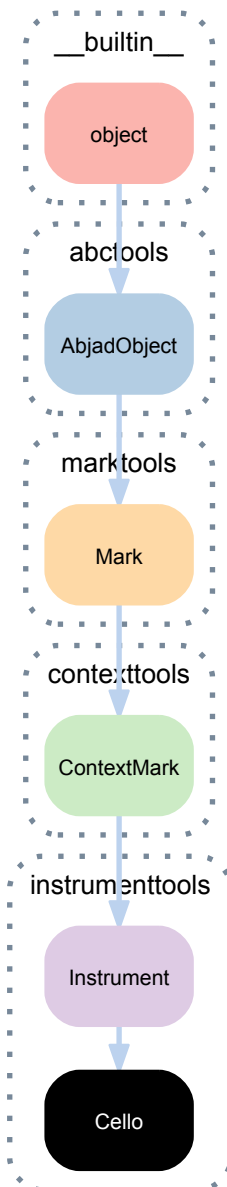
(Instrument) .**\_\_repr\_\_** ()

Interpreter representation of instrument.

Returns string.

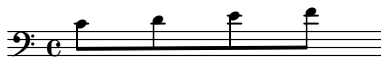


### 6.1.14 instrumenttools.Cello



**class** instrumenttools.**Cello** (\*\*kwargs)  
A cello.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> clef = contexttools.ClefMark('bass')
>>> clef = clef.attach(staff)
>>> show(staff)
```



```
>>> cello = instrumenttools.Cello()
>>> cello = cello.attach(staff)
>>> show(staff)
```



The cello targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument).**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

(Instrument).**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

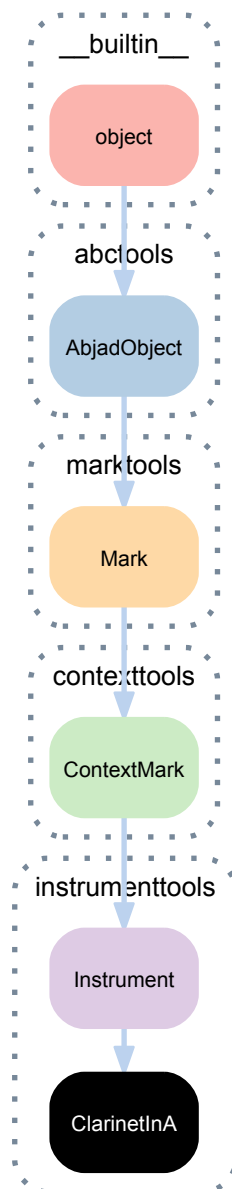
Returns boolean.

(Instrument).**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.15 instrumenttools.ClarinetInA



**class** instrumenttools.**ClarinetInA** (\*\*kwargs)  
A clarinet in A.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ClarinetInA()(staff)
ClarinetInA() (Staff{4})
```

```
>>> show(staff)
```

Clarinet in A 

The clarinet in A targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

### Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\**args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(*arg*)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

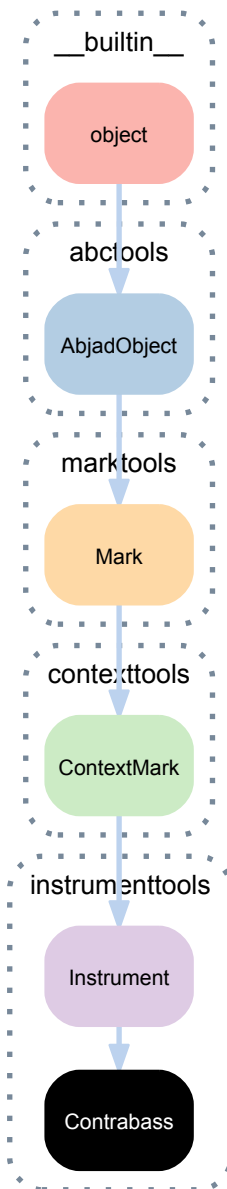
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

## 6.1.16 instrumenttools.Contrabass



**class** instrumenttools.**Contrabass** (\*\*kwargs)  
A contrabass.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.Contrabass()(staff)
Contrabass()(Staff{4})
```

```
>>> show(staff)
```

Contrabass

The contrabass targets staff context by default.

### Bases

- `instrumenttools.Instrument`

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.



(Instrument).**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

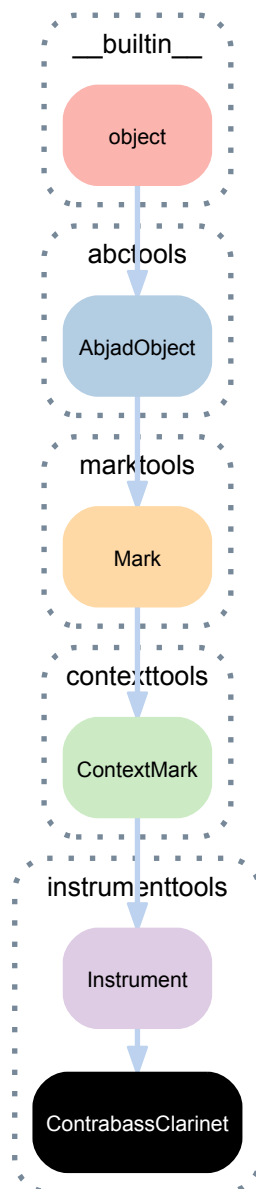
Returns boolean.

(Instrument).**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

## 6.1.17 instrumenttools.ContrabassClarinet



**class** instrumenttools.**ContrabassClarinet** (*\*\*kwargs*)  
 A contrabass clarinet.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ContrabassClarinet()(staff)
ContrabassClarinet()(Staff{4})
```

```
>>> show(staff)
```

Contrabass clarinet

The contrabass clarinet targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\**args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(*arg*)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

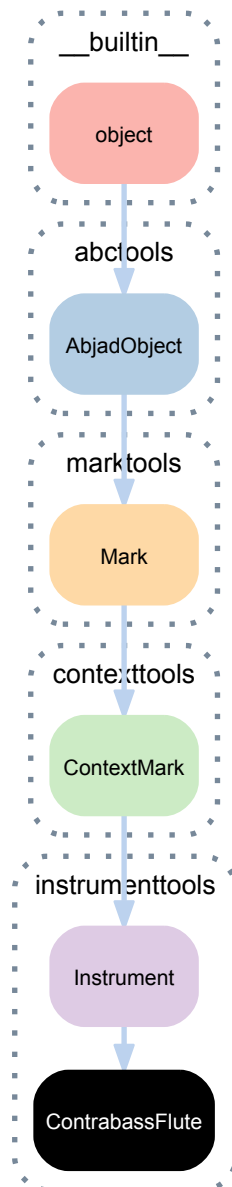
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.18 instrumenttools.ContrabassFlute



**class** instrumenttools.**ContrabassFlute** (*\*\*kwargs*)  
 A contrabass flute.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ContrabassFlute()(staff)
ContrabassFlute()(Staff(4))
```

```
>>> show(staff)
```

Contrabass flute 

The contrabass flute targets staff context by default.

#### Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

### Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

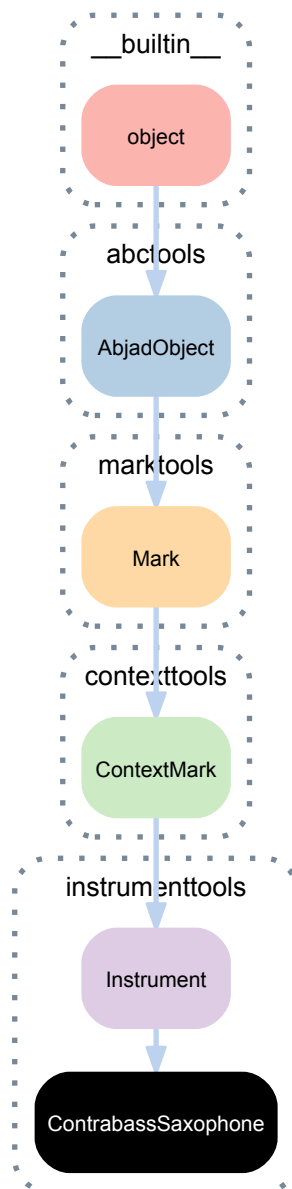
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.19 instrumenttools.ContrabassSaxophone



**class** instrumenttools.**ContrabassSaxophone** (\*\*kwargs)  
A bass saxophone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ContrabassSaxophone()(staff)
ContrabassSaxophone()(Staff{4})
```

```
>>> show(staff)
```

Contrabass saxophone 

The contrabass saxophone is pitched in E-flat.

The contrabass saxophone targets staff context by default.

#### Bases

- instrumenttools.Instrument



- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.  
Returns named pitch.

## Methods

`(ContextMark).attach(start_component)`  
Attaches context mark to *start\_component*.  
Makes sure no context mark of same type is already attached to score component that starts with start component.  
Returns context mark.

`(ContextMark).detach()`  
Detaches context mark.  
Returns context mark.

## Special methods

`(Mark).__call__(*args)`  
Detaches mark from component when called with no arguments.  
Attaches mark to component when called with one argument.  
Returns self.

`(Instrument).__copy__(*args)`  
Copies instrument.  
Returns new instrument.

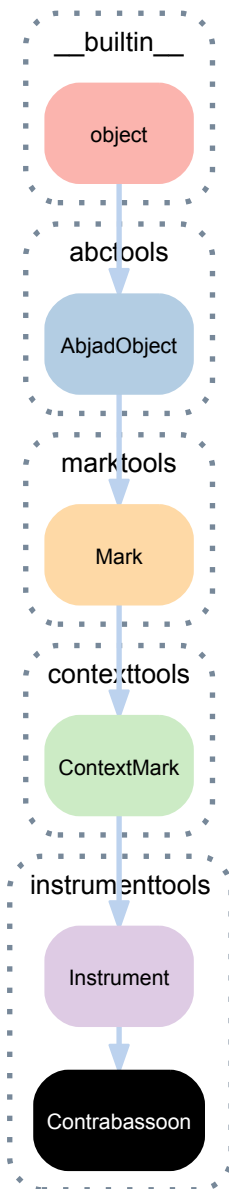
`(Instrument).__eq__(arg)`  
True when instrument equals *arg*. Otherwise false.  
Returns boolean.

`(Instrument).__hash__()`  
Hash value of instrument.  
Returns integer.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(Instrument).__repr__()`  
Interpreter representation of instrument.  
Returns string.

## 6.1.20 instrumenttools.Contrabassoon



**class** instrumenttools.**Contrabassoon** (\*\*kwargs)  
A contrabassoon.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.Contrabassoon()(staff)
Contrabassoon()(Staff{4})
```

```
>>> show(staff)
```

Contrabassoon 

The contrabassoon targets staff context by default.

### Bases

- `instrumenttools.Instrument`

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument) **.short\_instrument\_name\_markup**  
Gets and sets short instrument name markup.

Returns markup.

(Instrument).**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

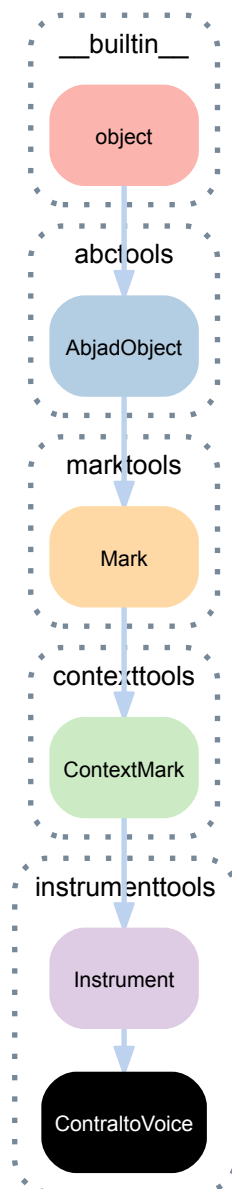
Returns boolean.

(Instrument).**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.21 instrumenttools.ContraltoVoice



**class** instrumenttools.**ContraltoVoice** (*\*\*kwargs*)  
 A contralto voice.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ContraltoVoice()(staff)
ContraltoVoice()(Staff{4})
```

```
>>> show(staff)
```

Contralto voice

The contralto voice targets staff context by default.

#### Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\**args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(*arg*)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

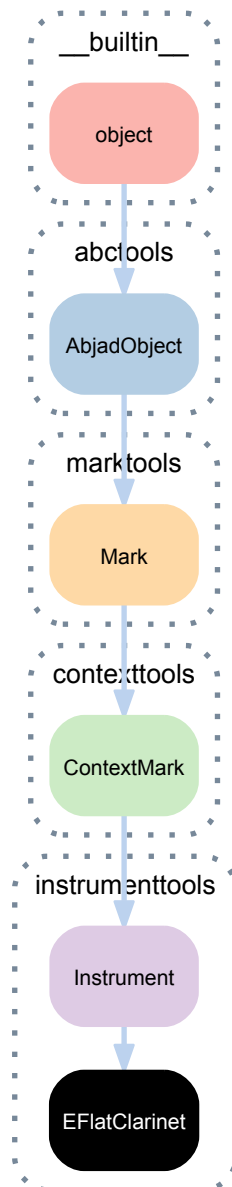
(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.



### 6.1.22 instrumenttools.EFlatClarinet



**class** instrumenttools.**EFlatClarinet** (\*\*kwargs)  
A E-flat clarinet.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.EFlatClarinet()(staff)
EFlatClarinet()(Staff{4})
```

```
>>> show(staff)
```

Clarinet in E-flat 

The E-flat clarinet targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

### Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument) **.short\_instrument\_name\_markup**  
Gets and sets short instrument name markup.

Returns markup.

(Instrument) **.sounding\_pitch\_of\_written\_middle\_c**  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

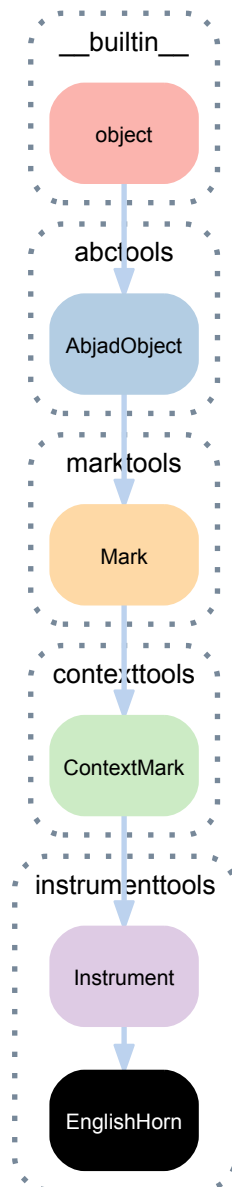
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.23 instrumenttools.EnglishHorn



**class** instrumenttools.**EnglishHorn** (\*\*kwargs)  
A English horn.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.EnglishHorn()(staff)
EnglishHorn()(Staff{4})
```

```
>>> show(staff)
```

English horn

The English horn targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

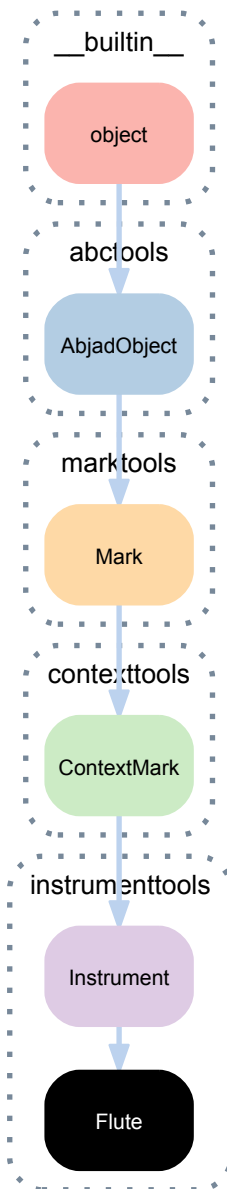
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.24 instrumenttools.Flute



**class** instrumenttools.**Flute** (\*\*kwargs)  
A flute.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> flute = instrumenttools.Flute()
>>> flute = flute.attach(staff)
>>> show(staff)
```



The flute targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**

Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**

LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**

Start component of mark.

Returns component or none.

(Mark) **.storage\_format**

Storage format of mark.

Returns string.

(ContextMark) **.target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**

Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**

Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**

Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**

Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**

Gets and sets short instrument name.

Returns string.



(Instrument).**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

(Instrument).**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

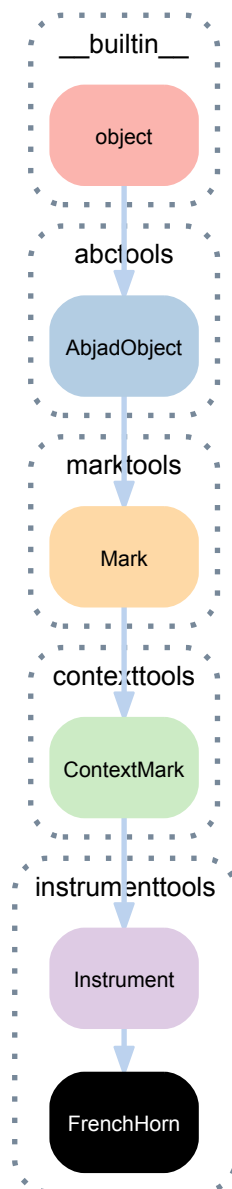
Returns boolean.

(Instrument).**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

## 6.1.25 instrumenttools.FrenchHorn



**class** instrumenttools.**FrenchHorn** (*\*\*kwargs*)  
 A French horn.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> french_horn = instrumenttools.FrenchHorn()
>>> french_horn = french_horn.attach(staff)
>>> show(staff)
```



The French horn targets staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**

Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**

LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**

Start component of mark.

Returns component or none.

(Mark) **.storage\_format**

Storage format of mark.

Returns string.

(ContextMark) **.target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**

Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**

Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**

Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**

Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**

Gets and sets short instrument name.

Returns string.

(Instrument) .**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

(Instrument) .**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

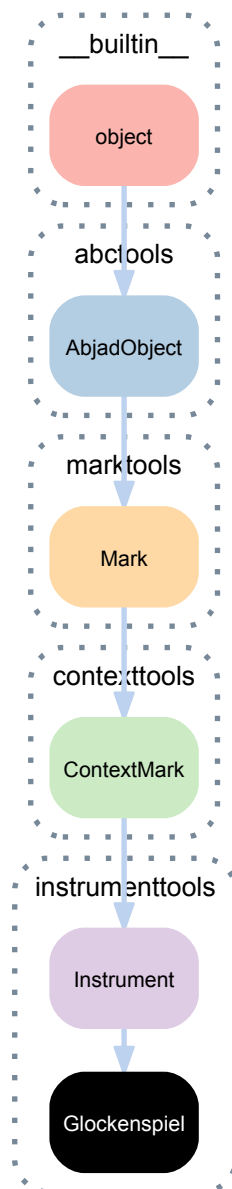
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.26 instrumenttools.Glockenspiel



**class** instrumenttools.**Glockenspiel** (*\*\*kwargs*)  
 A glockenspiel.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Glockenspiel() (staff)
Glockenspiel() (Staff{4})
```

```
>>> show(staff)
```

Glockenspiel

The glockenspiel targets staff context by default.

#### Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

### Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) **.attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) **.\_\_call\_\_**(\**args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) **.\_\_copy\_\_**(\**args*)

Copies instrument.

Returns new instrument.

(Instrument) **.\_\_eq\_\_**(*arg*)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) **.\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) **.\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

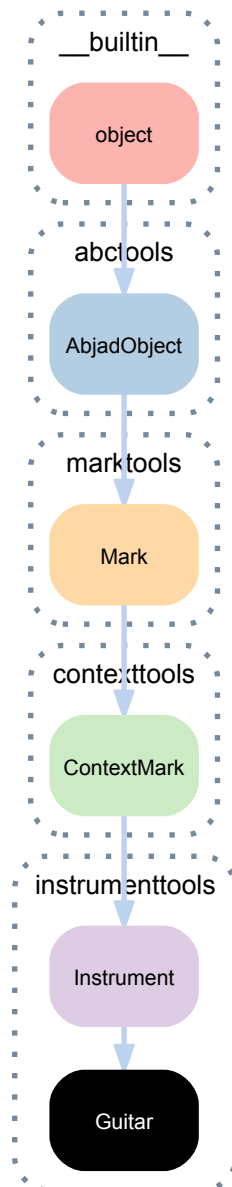
Returns boolean.

(Instrument) **.\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

## 6.1.27 instrumenttools.Guitar



**class** instrumenttools.**Guitar** (\*\*kwargs)  
A guitar.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Guitar()(staff)
Guitar() (Staff{4})
```

```
>>> show(staff)
```



The guitar targets staff context by default.

### Bases

- instrumenttools.Instrument
- contexttools.ContextMark



- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

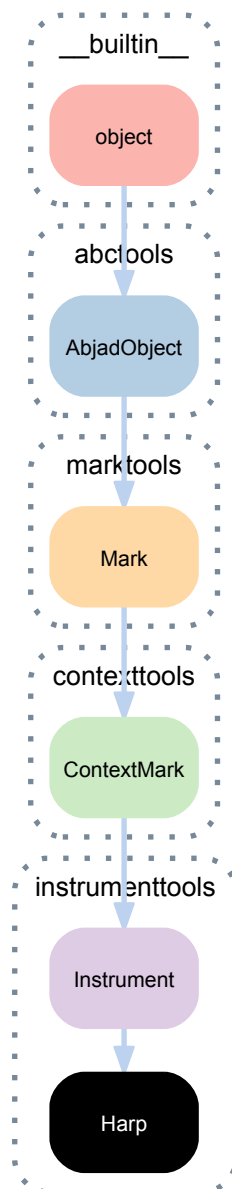
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.28 instrumenttools.Harp



**class** instrumenttools.**Harp** (*target\_context=None*, *\*\*kwargs*)  
 A harp.

```
>>> piano_staff = scoretools.PianoStaff([Staff("c'8 d'8 e'8 f'8"), Staff("c'4 b4")])
```

```
>>> instrumenttools.Harp()(piano_staff)
Harp() (PianoStaff<<2>>)
```

```
>>> show(piano_staff)
```



The harp targets piano staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument).**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

(Instrument).**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

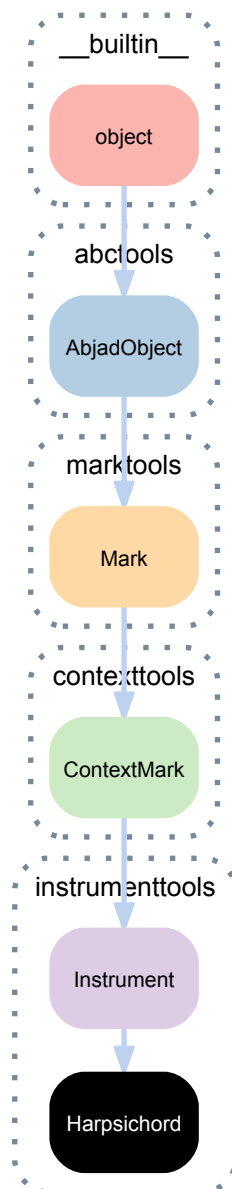
Returns boolean.

(Instrument).**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

## 6.1.29 instrumenttools.Harpsichord



**class** instrumenttools.**Harpsichord** (*target\_context=None*, *\*\*kwargs*)  
 A harpsichord.

```
>>> upper_staff = Staff("c'8 d'8 e'8 f'8")
>>> lower_staff = Staff("c'4 b4")
>>> piano_staff = scoretools.PianoStaff([upper_staff, lower_staff])
```

```
>>> instrumenttools.Harpsichord() (piano_staff)
Harpsichord() (PianoStaff<<2>>)
```

```
>>> show(piano_staff)
```



The harpsichord targets piano staff context by default.

Returns instrument.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**

Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**

LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**

Start component of mark.

Returns component or none.

(Mark) **.storage\_format**

Storage format of mark.

Returns string.

(ContextMark) **.target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**

Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**

Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**

Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**

Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**

Gets and sets short instrument name.

Returns string.

(Instrument) .**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

(Instrument) .**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

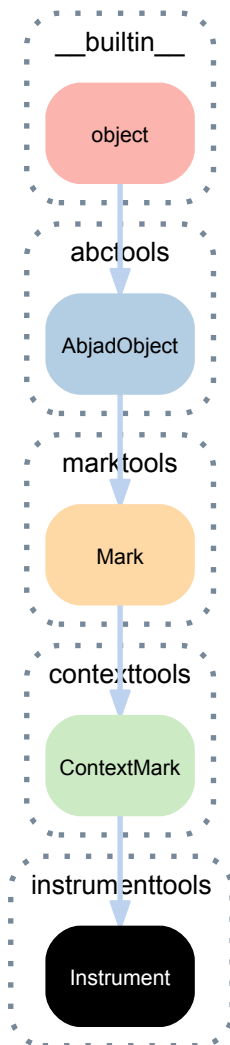
(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.



### 6.1.30 instrumenttools.Instrument



**class** `instrumenttools.Instrument` (*instrument\_name=None, short\_instrument\_name=None, instrument\_name\_markup=None, short\_instrument\_name\_markup=None, tar-get\_context=None*)

A musical instrument.

#### Bases

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

(`ContextMark`).**effective\_context**

Effective context of context mark.

Returns context mark or none.

`Instrument.lilypond_format`

LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`

Start component of mark.

Returns component or none.

`(Mark).storage_format`

Storage format of mark.

Returns string.

`(ContextMark).target_context`

Target context of context mark.

Returns context or none.

## Read/write properties

`Instrument.allowable_clefs`

Gets and sets allowable clefs.

Returns clef inventory.

`Instrument.instrument_name`

Gets and sets instrument name.

Returns string.

`Instrument.instrument_name_markup`

Gets and sets instrument name markup.

Returns markup.

`Instrument.pitch_range`

Gets and sets pitch range.

Returns pitch range.

`Instrument.short_instrument_name`

Gets and sets short instrument name.

Returns string.

`Instrument.short_instrument_name_markup`

Gets and sets short instrument name markup.

Returns markup.

`Instrument.sounding_pitch_of_written_middle_c`

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

`(ContextMark).attach(start_component)`

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

`(ContextMark).detach()`  
Detaches context mark.  
Returns context mark.

### Special methods

`(Mark).__call__(*args)`  
Detaches mark from component when called with no arguments.  
Attaches mark to component when called with one argument.  
Returns self.

`Instrument.__copy__(*args)`  
Copies instrument.  
Returns new instrument.

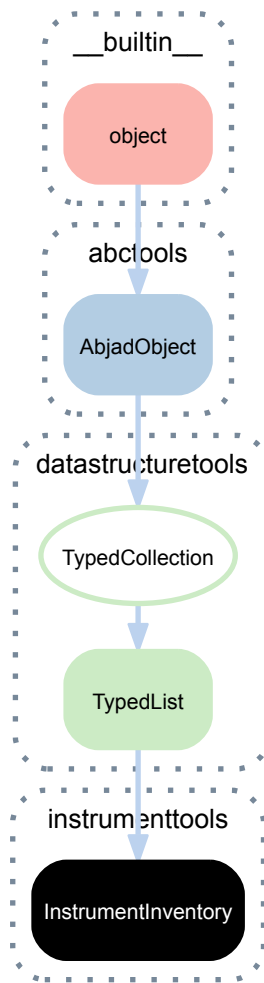
`Instrument.__eq__(arg)`  
True when instrument equals *arg*. Otherwise false.  
Returns boolean.

`Instrument.__hash__()`  
Hash value of instrument.  
Returns integer.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`Instrument.__repr__()`  
Interpreter representation of instrument.  
Returns string.

### 6.1.31 instrumenttools.InstrumentInventory



**class** `instrumenttools.InstrumentInventory` (*tokens=None, item\_class=None, name=None*)  
 An ordered list of instruments.

```
>>> inventory = instrumenttools.InstrumentInventory(
...     [instrumenttools.Flute(), instrumenttools.Guitar()])
```

```
>>> inventory
InstrumentInventory([Flute(), Guitar()])
```

Instrument inventories implement list interface and are mutable.

#### Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

(`TypedCollection`).**item\_class**  
 Item class to coerce tokens into.

`(TypedCollection) .storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection) .name`  
Read / write name of typed tuple.

## Methods

`(TypedList) .append(token)`  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

`(TypedList) .count(token)`  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)
3
```

Returns count.

`(TypedList) .extend(tokens)`  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

`(TypedList) .index(token)`  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('c'qf', "as'", 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

`(TypedList) .insert(i, token)`  
Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['1', 2, 4.3])
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

(TypedList) **.pop** (*i=-1*)

Aliases list.pop().

(TypedList) **.remove** (*token*)

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['0', 1.0, 2, 3.14159])
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()

Aliases list.reverse().

(TypedList) **.sort** (*cmp=None, key=None, reverse=False*)

Aliases list.sort().

## Special methods

(TypedCollection) **.\_\_contains\_\_** (*token*)

(TypedList) **.\_\_delitem\_\_** (*i*)

Aliases list.\_\_delitem\_\_().

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedList) **.\_\_getitem\_\_** (*i*)

Aliases list.\_\_getitem\_\_().

(TypedList) **.\_\_iadd\_\_** (*expr*)

Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
        'ppp',
        target_context=stafftools.Staff
    ),
```

```

contexttools.DynamicMark (
    'p',
    target_context=stafftools.Staff
),
contexttools.DynamicMark (
    'mp',
    target_context=stafftools.Staff
),
contexttools.DynamicMark (
    'mf',
    target_context=stafftools.Staff
),
contexttools.DynamicMark (
    'fff',
    target_context=stafftools.Staff
)
],
item_class=contexttools.DynamicMark
)

```

Returns collection.

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

(TypedCollection).**\_\_ne\_\_**(*expr*)

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList).**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList).**\_\_setitem\_\_**(*i*, *expr*)

Change tokens in *expr* to items and set:

```

>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch('gqs,')
],
item_class=pitchtools.NamedPitch
)

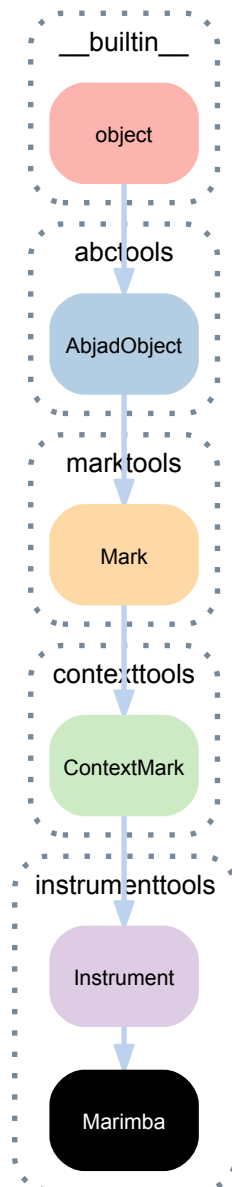
```

```

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch("f'"),
    pitchtools.NamedPitch("g'"),
    pitchtools.NamedPitch("a'"),
    pitchtools.NamedPitch("b'"),
    pitchtools.NamedPitch("c'")
],
item_class=pitchtools.NamedPitch
)

```

### 6.1.32 instrumenttools.Marimba



**class** instrumenttools.**Marimba** (\*\*kwargs)  
A marimba.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Marimba()(staff)
Marimba()(Staff{4})
```

```
>>> show(staff)
```



The marimba targets staff context by default.

#### Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`



- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

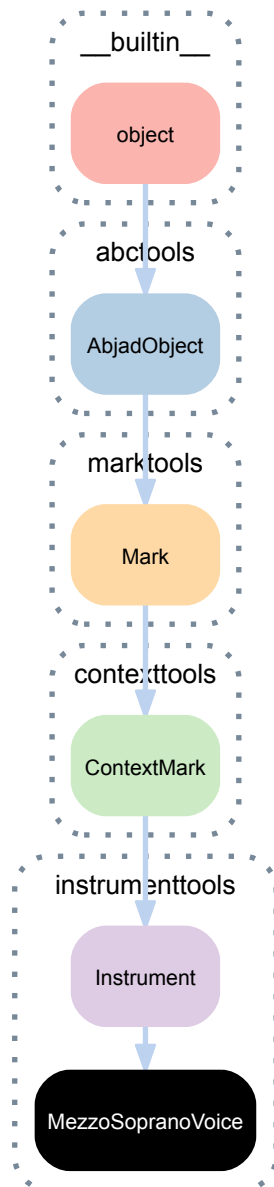
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.33 instrumenttools.MezzoSopranoVoice



**class** instrumenttools.**MezzoSopranoVoice** (\*\*kwargs)  
A mezzo-soprano voice.

```
>>> staff = Staff("c''8 d''8 e''8 f''8")
```

```
>>> instrumenttools.MezzoSopranoVoice()(staff)
MezzoSopranoVoice()(Staff{4})
```

```
>>> show(staff)
```

Mezzo-soprano voice 

The mezzo-soprano voice targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

### Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

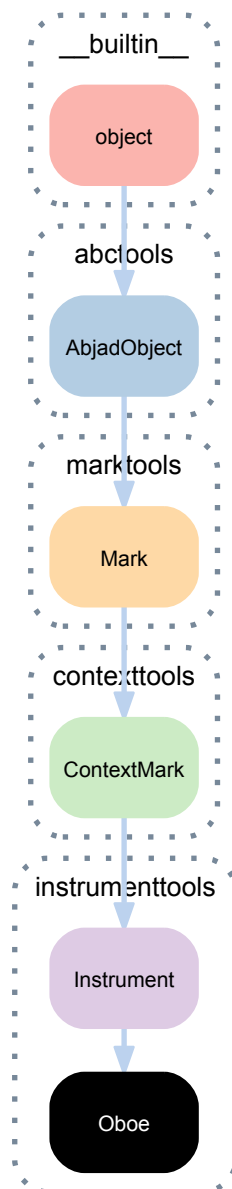
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.34 instrumenttools.Oboe



**class** instrumenttools.Oboe (\*\*kwargs)  
An oboe.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Oboe()(staff)
Oboe()(Staff{4})
```

```
>>> show(staff)
```



The oboe targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) **.attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) **.\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) **.\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) **.\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) **.\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

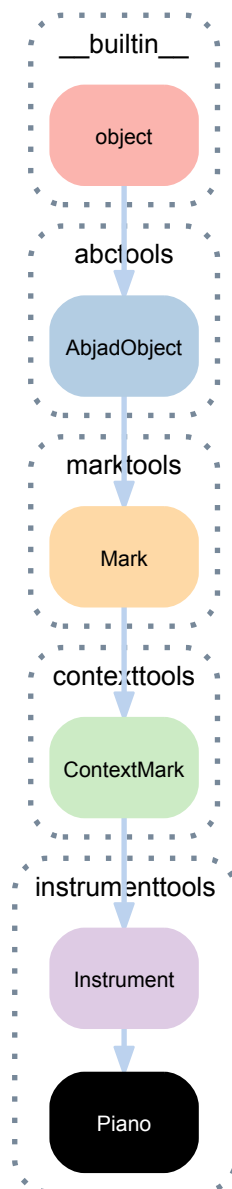
(Instrument) **.\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.



### 6.1.35 instrumenttools.Piano



**class** instrumenttools.**Piano** (*target\_context=None*, *\*\*kwargs*)  
A piano.

```
>>> piano_staff = scoretools.PianoStaff(
...     [Staff("c'8 d'8 e'8 f'8"), Staff("c'4 b4")])
```

```
>>> instrumenttools.Piano()(piano_staff)
Piano()(PianoStaff<<2>>)
```

```
>>> show(piano_staff)
```



The piano targets piano staff context by default.

## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument).**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

(Instrument).**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

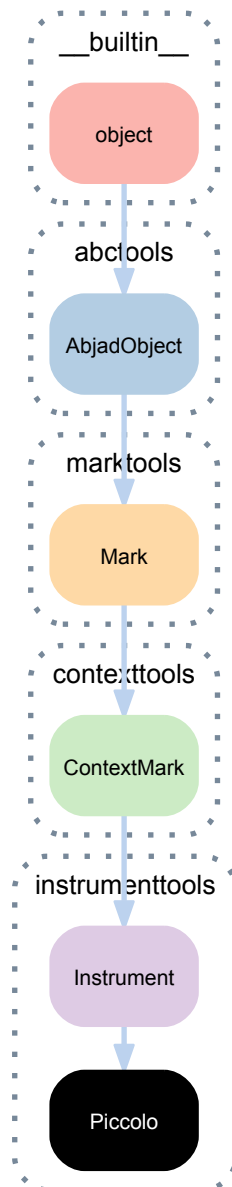
Returns boolean.

(Instrument).**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.36 instrumenttools.Piccolo



**class** instrumenttools.**Piccolo**(\*\*kwargs)  
A piccolo.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Piccolo()(staff)
Piccolo()(Staff{4})
```

```
>>> show(staff)
```

Piccolo 

The piccolo targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\**args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(*arg*)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

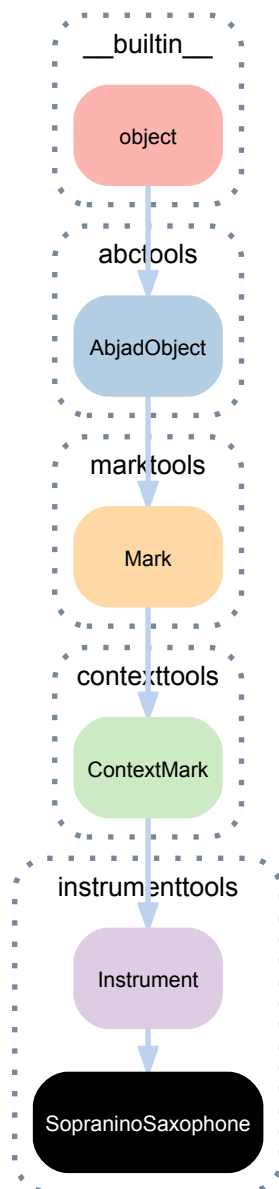
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.37 instrumenttools.SopraninoSaxophone



**class** instrumenttools.**SopraninoSaxophone** (\*\*kwargs)  
A sopranino saxophone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.SopraninoSaxophone()(staff)
SopraninoSaxophone()(Staff{4})
```

```
>>> show(staff)
```

Sopranino saxophone 

The sopranino saxophone is pitched in E-flat.

The sopranino saxophone targets staff context by default.

#### Bases

- instrumenttools.Instrument

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.



`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.  
Returns named pitch.

## Methods

`(ContextMark).attach(start_component)`  
Attaches context mark to *start\_component*.  
Makes sure no context mark of same type is already attached to score component that starts with start component.  
Returns context mark.

`(ContextMark).detach()`  
Detaches context mark.  
Returns context mark.

## Special methods

`(Mark).__call__(*args)`  
Detaches mark from component when called with no arguments.  
Attaches mark to component when called with one argument.  
Returns self.

`(Instrument).__copy__(*args)`  
Copies instrument.  
Returns new instrument.

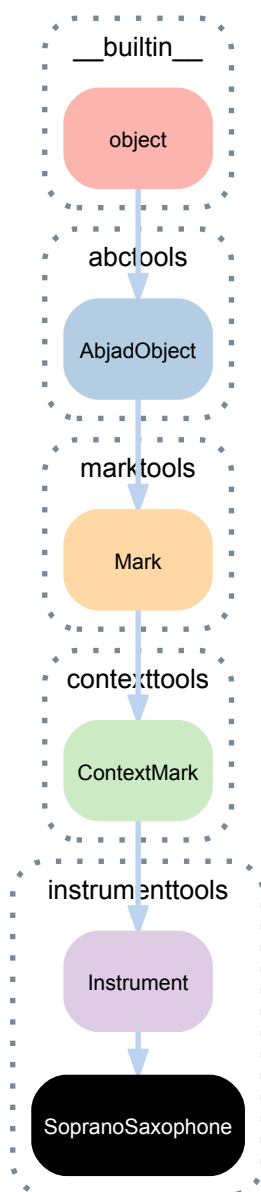
`(Instrument).__eq__(arg)`  
True when instrument equals *arg*. Otherwise false.  
Returns boolean.

`(Instrument).__hash__()`  
Hash value of instrument.  
Returns integer.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(Instrument).__repr__()`  
Interpreter representation of instrument.  
Returns string.

### 6.1.38 instrumenttools.SopranoSaxophone



**class** instrumenttools.**SopranoSaxophone** (*\*\*kwargs*)  
A soprano saxophone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.SopranoSaxophone() (staff)
SopranoSaxophone() (Staff{4})
```

```
>>> show(staff)
```

Soprano saxophone

The soprano saxophone is pitched in B-flat.

The soprano saxophone targets staff context by default.

#### Bases

- `instrumenttools.Instrument`

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

(Instrument) .**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach** (*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach** ()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_** (\**args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_** (\**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_** (*arg*)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_** ()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

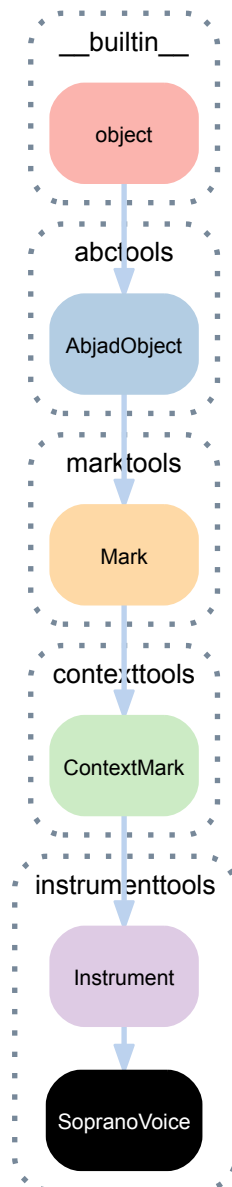
Returns boolean.

(Instrument) .**\_\_repr\_\_** ()

Interpreter representation of instrument.

Returns string.

### 6.1.39 instrumenttools.SopranoVoice



**class** instrumenttools.**SopranoVoice** (*\*\*kwargs*)  
 A soprano voice.

```
>>> staff = Staff("c''8 d''8 e''8 f''8")
```

```
>>> instrumenttools.SopranoVoice()(staff)
SopranoVoice()(Staff{4})
```

```
>>> show(staff)
```

Soprano voice 

The soprano voice targets staff context by default.

#### Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

### Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\**args*)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(*arg*)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

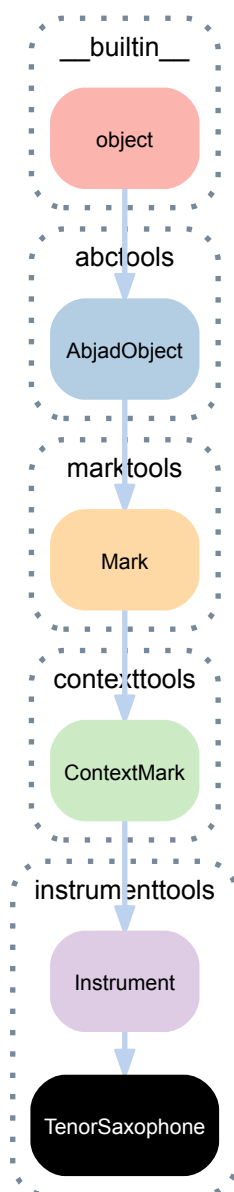
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.40 instrumenttools.TenorSaxophone



**class** instrumenttools.TenorSaxophone (\*\*kwargs)  
A tenor saxophone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.TenorSaxophone()(staff)
TenorSaxophone()(Staff{4})
```

```
>>> show(staff)
```

Tenor saxophone

The tenor saxophone targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark



- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

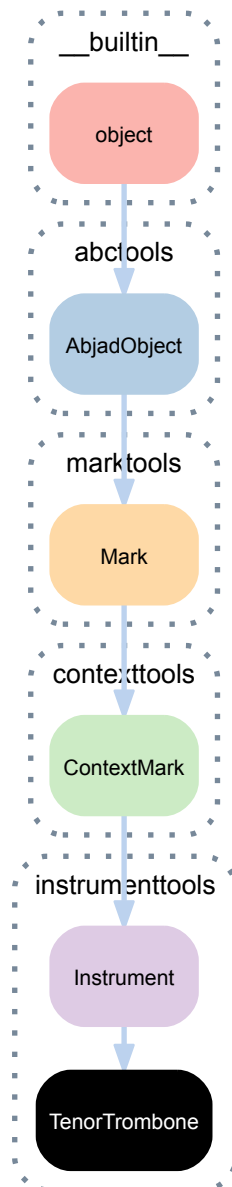
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.41 instrumenttools.TenorTrombone



**class** instrumenttools.TenorTrombone (\*\*kwargs)  
A tenor trombone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.TenorTrombone()(staff)
TenorTrombone()(Staff{4})
```

```
>>> show(staff)
```

Tenor trombone 

The tenor trombone targets staff context by default.

#### Bases

- instrumenttools.Instrument

- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

(Instrument).**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark).**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark).**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark).**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument).**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument).**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument).**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

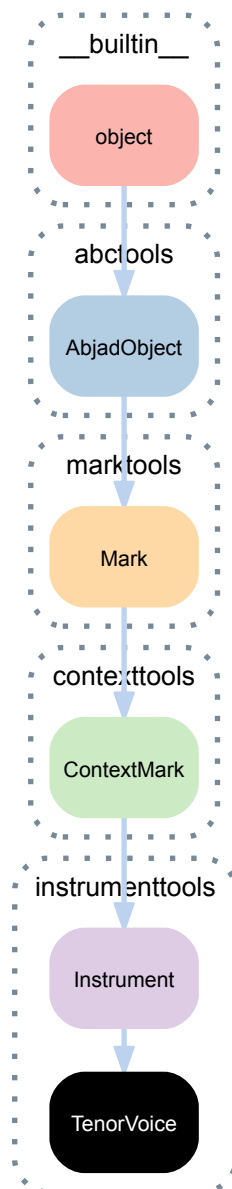
Returns boolean.

(Instrument).**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

## 6.1.42 instrumenttools.TenorVoice



**class** instrumenttools.**TenorVoice** (*\*\*kwargs*)  
 A tenor voice.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.TenorVoice()(staff)
TenorVoice()(Staff{4})
```

```
>>> show(staff)
```

Tenor voice 

The tenor voice targets staff context by default.

### Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(ContextMark).effective_context`  
Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`  
LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

`(ContextMark).target_context`  
Target context of context mark.

Returns context or none.

## Read/write properties

`(Instrument).allowable_clefs`  
Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`  
Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`  
Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`  
Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`  
Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`  
Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

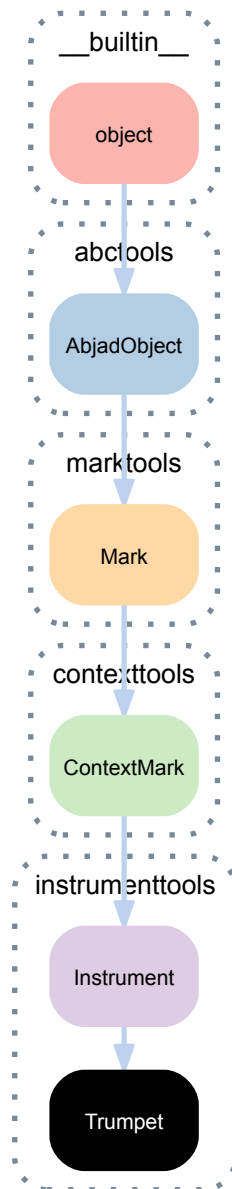
(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.



### 6.1.43 instrumenttools.Trumpet



**class** instrumenttools.**Trumpet** (\*\*kwargs)  
A trumpet.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Trumpet() (staff)
Trumpet() (Staff{4})
```

The trumpet targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark
- marktools.Mark
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument) **.short\_instrument\_name\_markup**  
Gets and sets short instrument name markup.

Returns markup.

(Instrument) **.sounding\_pitch\_of\_written\_middle\_c**  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

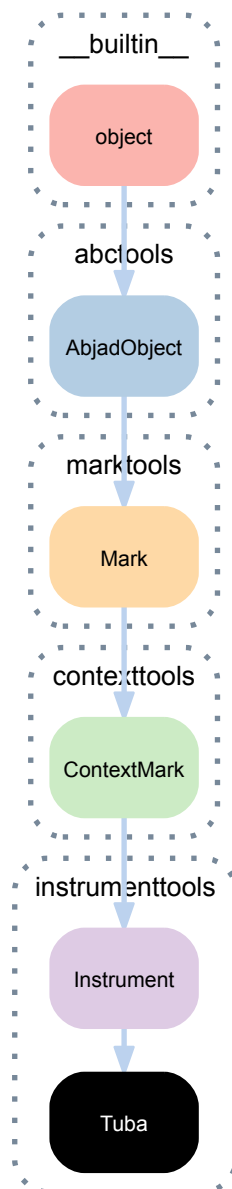
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.44 instrumenttools.Tuba



**class** instrumenttools.Tuba (\*\*kwargs)  
A tuba.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.Tuba()(staff)
Tuba()(Staff{4})
```

The tuba targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark
- marktools.Mark

- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`

Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`

LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`

Start component of mark.

Returns component or none.

`(Mark).storage_format`

Storage format of mark.

Returns string.

`(ContextMark).target_context`

Target context of context mark.

Returns context or none.

### Read/write properties

`(Instrument).allowable_clefs`

Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`

Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`

Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`

Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`

Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`

Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

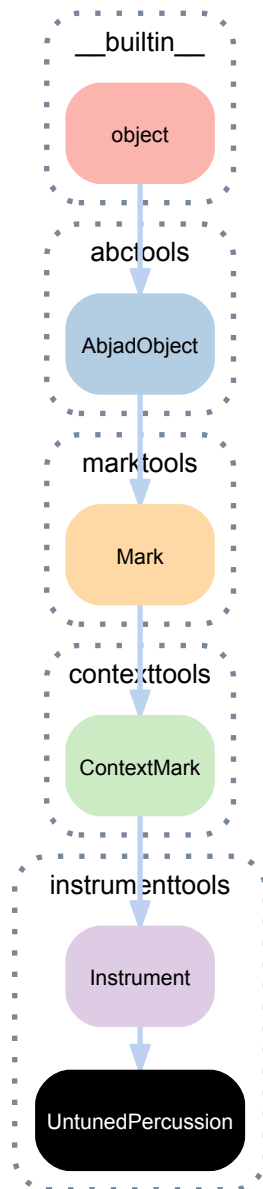
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.45 instrumenttools.UntunedPercussion



**class** instrumenttools.UntunedPercussion (\*\*kwargs)  
An untuned percussion instrument.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.UntunedPercussion()(staff)
UntunedPercussion()(Staff{4})
```

Untuned percussion targets the staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark
- marktools.Mark
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument) **.short\_instrument\_name\_markup**  
Gets and sets short instrument name markup.

Returns markup.

(Instrument) **.sounding\_pitch\_of\_written\_middle\_c**  
Gets and sets sounding pitch of written middle C.

Returns named pitch.



## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

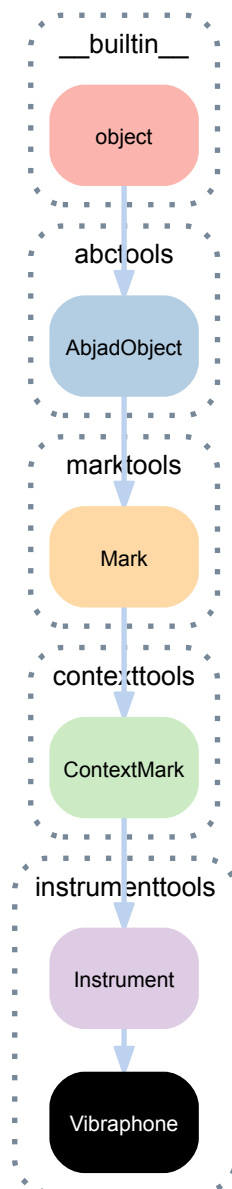
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.46 instrumenttools.Vibraphone



```
class instrumenttools.Vibraphone (**kwargs)
    A vibraphone.
```

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Vibraphone()(staff)
Vibraphone()(Staff{4})
```

The vibraphone targets staff context by default.

#### Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument) **.short\_instrument\_name\_markup**  
Gets and sets short instrument name markup.

Returns markup.

(Instrument) **.sounding\_pitch\_of\_written\_middle\_c**  
Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

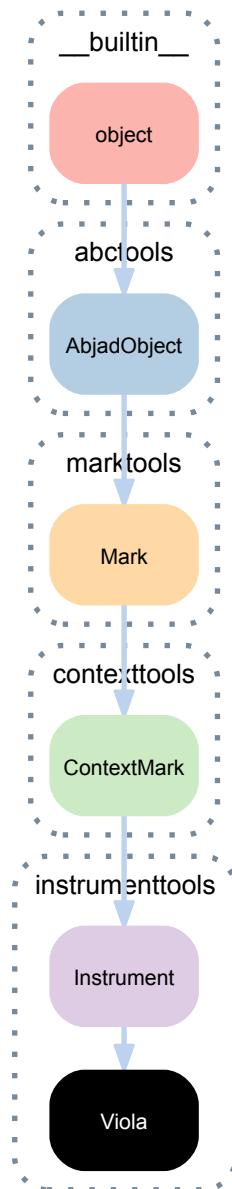
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.47 instrumenttools.Viola



**class** instrumenttools.**Viola** (\*\*kwargs)  
A viola.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('alto')(staff)
ClefMark('alto')(Staff{4})
```

```
>>> instrumenttools.Viola()(staff)
Viola()(Staff{4})
```

The viola targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark
- marktools.Mark

- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(ContextMark).effective_context`

Effective context of context mark.

Returns context mark or none.

`(Instrument).lilypond_format`

LilyPond format of instrument mark.

Returns string.

`(Mark).start_component`

Start component of mark.

Returns component or none.

`(Mark).storage_format`

Storage format of mark.

Returns string.

`(ContextMark).target_context`

Target context of context mark.

Returns context or none.

### Read/write properties

`(Instrument).allowable_clefs`

Gets and sets allowable clefs.

Returns clef inventory.

`(Instrument).instrument_name`

Gets and sets instrument name.

Returns string.

`(Instrument).instrument_name_markup`

Gets and sets instrument name markup.

Returns markup.

`(Instrument).pitch_range`

Gets and sets pitch range.

Returns pitch range.

`(Instrument).short_instrument_name`

Gets and sets short instrument name.

Returns string.

`(Instrument).short_instrument_name_markup`

Gets and sets short instrument name markup.

Returns markup.

`(Instrument).sounding_pitch_of_written_middle_c`

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

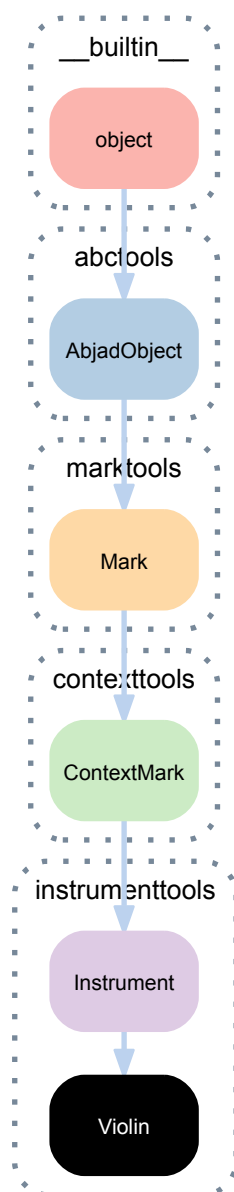
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

## 6.1.48 instrumenttools.Violin



**class** instrumenttools.**Violin** (\*\*kwargs)  
A violin.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> violin = instrumenttools.Violin()
>>> violin = violin.attach(staff)
>>> show(staff)
```



The violin targets staff context by default.



## Bases

- `instrumenttools.Instrument`
- `contexttools.ContextMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(ContextMark) **.effective\_context**

Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**

LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**

Start component of mark.

Returns component or none.

(Mark) **.storage\_format**

Storage format of mark.

Returns string.

(ContextMark) **.target\_context**

Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**

Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**

Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**

Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**

Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**

Gets and sets short instrument name.

Returns string.

(Instrument) .**short\_instrument\_name\_markup**

Gets and sets short instrument name markup.

Returns markup.

(Instrument) .**sounding\_pitch\_of\_written\_middle\_c**

Gets and sets sounding pitch of written middle C.

Returns named pitch.

## Methods

(ContextMark) .**attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with start component.

Returns context mark.

(ContextMark) .**detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) .**\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) .**\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) .**\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) .**\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

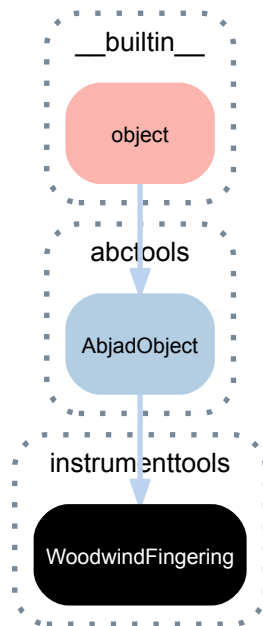
Returns boolean.

(Instrument) .**\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

### 6.1.49 instrumenttools.WoodwindFingering



**class** `instrumenttools.WoodwindFingering` (*arg*, *center\_column=None*, *left\_hand=None*, *right\_hand=None*)

A woodwind fingering.

Initialize from a valid instrument name and up to three keyword lists or tuples:

```
>>> center_column = ('one', 'two', 'three', 'five')
>>> left_hand = ('R', 'thumb')
>>> right_hand = ('e',)
>>> woodwind_fingering = instrumenttools.WoodwindFingering(
...     'clarinet',
...     center_column=center_column,
...     left_hand=left_hand,
...     right_hand=right_hand,
... )
```

```
>>> print woodwind_fingering.storage_format
instrumenttools.WoodwindFingering(
    'clarinet',
    center_column=('one', 'two', 'three', 'five'),
    left_hand=('R', 'thumb'),
    right_hand=('e',)
)
```

Initialize a `WoodwindFingering` from another `WoodwindFingering`:

```
>>> woodwind_fingering_2 = instrumenttools.WoodwindFingering(
...     woodwind_fingering)
>>> print woodwind_fingering_2.storage_format
instrumenttools.WoodwindFingering(
    'clarinet',
    center_column=('one', 'two', 'three', 'five'),
    left_hand=('R', 'thumb'),
    right_hand=('e',)
)
```

Call a `WoodwindFingering` to create a woodwind diagram `MarkupCommand`:

```
>>> fingering_command = woodwind_fingering()
>>> print fingering_command.storage_format
markuptools.MarkupCommand(
    'woodwind-diagram',
    schemetools.Scheme('c', 'l', 'a', 'r', 'i', 'n', 'e', 't'),
    schemetools.Scheme(schemetools.SchemePair('cc', ('one', 'two', 'three', 'five'))),
)
```

```

schemetools.SchemePair('lh', ('R', 'thumb')),
schemetools.SchemePair('rh', ('e',)))
)

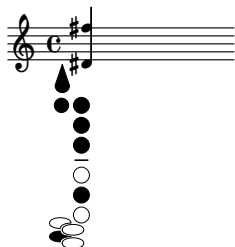
```

Attach the MarkupCommand to score components, such as a chord representing a multiphonic sound:

```

>>> markup = markuptools.Markup(fingering_command, direction=Down)
>>> chord = Chord("<ds' fs'>4")
>>> markup = markup.attach(chord)
>>> show(chord)

```



Initialize fingerings for eight different woodwind instruments:

```

>>> instrument_names = [
...     'piccolo', 'flute', 'oboe', 'clarinet', 'bass-clarinet',
...     'saxophone', 'bassoon', 'contrabassoon',
... ]
>>> for name in instrument_names:
...     instrumenttools.WoodwindFingering(name)
...
WoodwindFingering('piccolo', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('flute', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('oboe', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('clarinet', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('bass-clarinet', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('saxophone', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('bassoon', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('contrabassoon', center_column=(), left_hand=(), right_hand=())

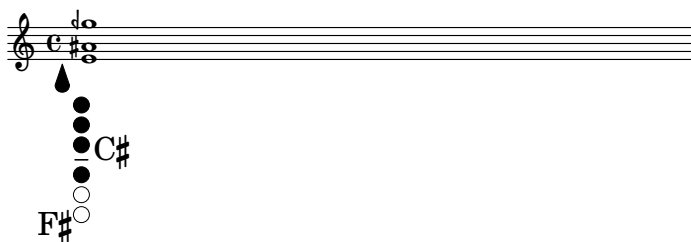
```

An override displays diagrams symbolically instead of graphically:

```

>>> chord = Chord("e' as' ggf'", (1,1))
>>> fingering = instrumenttools.WoodwindFingering(
...     'clarinet',
...     center_column=['one', 'two', 'three', 'four'],
...     left_hand=['R', 'cis'],
...     right_hand=['fis'])
>>> diagram = fingering()
>>> not_graphical = markuptools.MarkupCommand(
...     'override',
...     schemetools.SchemePair('graphical', False))
>>> markup = markuptools.Markup(
...     [not_graphical, diagram], direction=Down)
>>> markup = markup.attach(chord)
>>> show(chord)

```



The thickness and size of diagrams can also be changed with overrides:

```

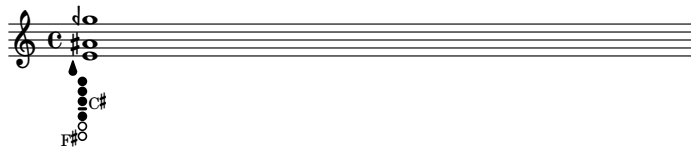
>>> chord = Chord("e' as' ggf'", (1,1))
>>> fingering = instrumenttools.WoodwindFingering(
...     'clarinet',

```

```

...     center_column=('one', 'two', 'three', 'four'),
...     left_hand=('R', 'cis'),
...     right_hand=('fis',),
... )
>>> diagram = fingering()
>>> not_graphical = markuptools.MarkupCommand(
...     'override',
...     schemetools.SchemePair('graphical', False))
>>> size = markuptools.MarkupCommand(
...     'override', schemetools.SchemePair('size', .5))
>>> thickness = markuptools.MarkupCommand(
...     'override', schemetools.SchemePair('thickness', .4))
>>> markup = markuptools.Markup(
...     [not_graphical, size, thickness, diagram], direction=Down)
>>> markup = markup.attach(chord)
>>> show(chord)

```



Inspired by Mike Solomon's LilyPond woodwind diagrams.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`WoodwindFingering.center_column`

**Tuple of contents of key strings in center column key group:**

```
>>> woodwind_fingering.center_column
('one', 'two', 'three', 'five')
```

Returns tuple.

`WoodwindFingering.instrument_name`

**String of valid woodwind instrument name:**

```
>>> woodwind_fingering.instrument_name
'clarinet'
```

Returns string.

`WoodwindFingering.left_hand`

**Tuple of contents of key strings in left hand key group:**

```
>>> woodwind_fingering.left_hand
('R', 'thumb')
```

Returns tuple.

`WoodwindFingering.right_hand`

**Tuple of contents of key strings in right hand key group:**

```
>>> woodwind_fingering.right_hand
('e',)
```

Returns tuple.

WoodwindFingering.**storage\_format**

Storage format of woodwind fingering.

Returns string.

## Methods

WoodwindFingering.**print\_guide**()

Print read-only string containing instrument's valid key strings, instrument diagram, and syntax explanation.

Returns string.

## Special methods

WoodwindFingering.**\_\_call\_\_**()

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

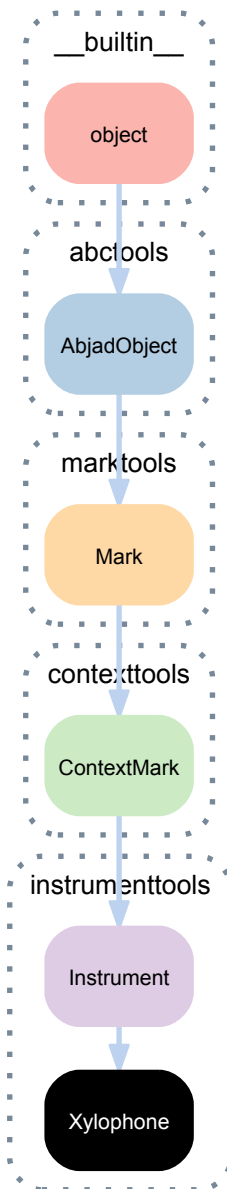
Returns boolean.

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

### 6.1.50 instrumenttools.Xylophone



**class** instrumenttools.**Xylophone** (\*\*kwargs)  
A xylophone.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Xylophone() (staff)
Xylophone() (Staff{4})
```

The xylophone targets staff context by default.

#### Bases

- instrumenttools.Instrument
- contexttools.ContextMark
- marktools.Mark
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

(ContextMark) **.effective\_context**  
Effective context of context mark.

Returns context mark or none.

(Instrument) **.lilypond\_format**  
LilyPond format of instrument mark.

Returns string.

(Mark) **.start\_component**  
Start component of mark.

Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.

Returns string.

(ContextMark) **.target\_context**  
Target context of context mark.

Returns context or none.

## Read/write properties

(Instrument) **.allowable\_clefs**  
Gets and sets allowable clefs.

Returns clef inventory.

(Instrument) **.instrument\_name**  
Gets and sets instrument name.

Returns string.

(Instrument) **.instrument\_name\_markup**  
Gets and sets instrument name markup.

Returns markup.

(Instrument) **.pitch\_range**  
Gets and sets pitch range.

Returns pitch range.

(Instrument) **.short\_instrument\_name**  
Gets and sets short instrument name.

Returns string.

(Instrument) **.short\_instrument\_name\_markup**  
Gets and sets short instrument name markup.

Returns markup.

(Instrument) **.sounding\_pitch\_of\_written\_middle\_c**  
Gets and sets sounding pitch of written middle C.

Returns named pitch.



## Methods

(ContextMark) **.attach**(*start\_component*)

Attaches context mark to *start\_component*.

Makes sure no context mark of same type is already attached to score component that starts with *start\_component*.

Returns context mark.

(ContextMark) **.detach**()

Detaches context mark.

Returns context mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

(Instrument) **.\_\_copy\_\_**(\*args)

Copies instrument.

Returns new instrument.

(Instrument) **.\_\_eq\_\_**(arg)

True when instrument equals *arg*. Otherwise false.

Returns boolean.

(Instrument) **.\_\_hash\_\_**()

Hash value of instrument.

Returns integer.

(AbjadObject) **.\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Instrument) **.\_\_repr\_\_**()

Interpreter representation of instrument.

Returns string.

## 6.2 Functions

### 6.2.1 instrumenttools.default\_instrument\_name\_to\_instrument\_class

`instrumenttools.default_instrument_name_to_instrument_class(default_instrument_name)`

Change *default\_instrument\_name* to class name:

```
>>> instrumenttools.default_instrument_name_to_instrument_class('clarinet in E-flat')
<class 'abjad.tools.instrumenttools.EFlatClarinet.EFlatClarinet.EFlatClarinet'>
```

Returns class.

When *default\_instrument\_name* matches no instrument class:

```
>>> instrumenttools.default_instrument_name_to_instrument_class('foo') is None
True
```

Returns none.

## 6.2.2 instrumenttools.iterate\_out\_of\_range\_notes\_and\_chords

`instrumenttools.iterate_out_of_range_notes_and_chords` (*expr*)

Iterates notes and chords in *expr* outside traditional instrument ranges:

```
>>> staff = Staff("c'8 r8 <d fs>8 r8")
>>> instrumenttools.Violin() (staff)
Violin() (Staff{4})
```

```
>>> list(
... instrumenttools.iterate_out_of_range_notes_and_chords(
... staff))
[Chord('<d fs>8')]
```

Returns generator.

## 6.2.3 instrumenttools.notes\_and\_chords\_are\_in\_range

`instrumenttools.notes_and_chords_are_in_range` (*expr*)

True when notes and chords in *expr* are within traditional instrument ranges.

```
>>> staff = Staff("c'8 r8 <d' fs'>8 r8")
>>> instrumenttools.Violin() (staff)
Violin() (Staff{4})
```

```
>>> instrumenttools.notes_and_chords_are_in_range(
... staff)
True
```

False otherwise:

```
>>> staff = Staff("c'8 r8 <d fs>8 r8")
>>> instrumenttools.Violin() (staff)
Violin() (Staff{4})
```

```
>>> instrumenttools.notes_and_chords_are_in_range(
... staff)
False
```

Returns boolean.

## 6.2.4 instrumenttools.notes\_and\_chords\_are\_on\_expected\_clefs

`instrumenttools.notes_and_chords_are_on_expected_clefs` (*expr*, *percussion\_clef\_is\_allowed=True*)

True when notes and chords in *expr* are on expected clefs.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble') (staff)
ClefMark('treble') (Staff{4})
>>> instrumenttools.Violin() (staff)
Violin() (Staff{4})
```

```
>>> instrumenttools.notes_and_chords_are_on_expected_clefs(staff)
True
```

False otherwise:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('alto') (staff)
ClefMark('alto') (Staff{4})
>>> instrumenttools.Violin() (staff)
Violin() (Staff{4})
```

```
>>> instrumenttools.notes_and_chords_are_on_expected_clefs(staff)
False
```

Allows percussion clef when *percussion\_clef\_is\_allowed* is true:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('percussion')(staff)
ClefMark('percussion')(Staff{4})
>>> instrumenttools.Violin()(staff)
Violin()(Staff{4})
```

```
>>> instrumenttools.notes_and_chords_are_on_expected_clefs(
...     staff, percussion_clef_is_allowed=True)
True
```

Disallows percussion clef when *percussion\_clef\_is\_allowed* is false:

```
>>> instrumenttools.notes_and_chords_are_on_expected_clefs(
...     staff, percussion_clef_is_allowed=False)
False
```

Returns boolean.

## 6.2.5 instrumenttools.transpose\_from\_sounding\_pitch\_to\_written\_pitch

`instrumenttools.transpose_from_sounding_pitch_to_written_pitch(expr)`

Transpose notes and chords in *expr* from sounding pitch to written pitch:

```
>>> staff = Staff("<c' e' g'>4 d'4 r4 e'4")
>>> instrumenttools.BFlatClarinet()(staff)
BFlatClarinet()(Staff{4})
```

```
>>> instrumenttools.transpose_from_sounding_pitch_to_written_pitch(staff)
```

Returns none.

## 6.2.6 instrumenttools.transpose\_from\_written\_pitch\_to\_sounding\_pitch

`instrumenttools.transpose_from_written_pitch_to_sounding_pitch(expr)`

Transpose notes and chords in *expr* from sounding pitch to written pitch:

```
>>> staff = Staff("<c' e' g'>4 d'4 r4 e'4")
>>> clarinet = instrumenttools.BFlatClarinet()
>>> clarinet = clarinet.attach(staff)
>>> show(staff)
```

Clarinet in B-flat 

```
>>> for leaf in staff.select_leaves():
...     leaf.written_pitch_indication_is_at_sounding_pitch = False
>>> instrumenttools.transpose_from_written_pitch_to_sounding_pitch(staff)
>>> show(staff)
```

Clarinet in B-flat 

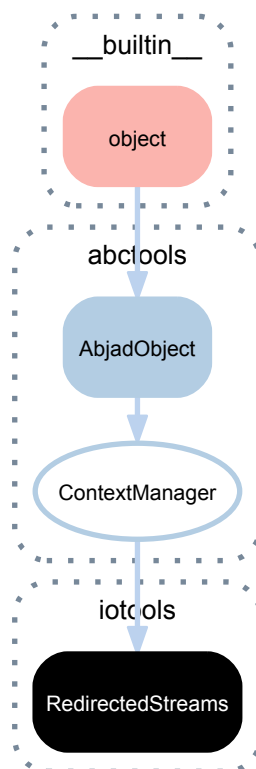
Returns none.



# IOTOOLS

## 7.1 Concrete classes

### 7.1.1 `iotools.RedirectedStreams`



**class** `iotools.RedirectedStreams` (*stdout=None, stderr=None*)  
A context manager for capturing stdout and stderr output.

```
>>> import StringIO
>>> string_io = StringIO.StringIO()
>>> with iotools.RedirectedStreams(stdout=string_io):
...     print "hello, world!"
...
>>> result = string_io.getvalue()
>>> string_io.close()
>>> print result
hello, world!
```

Returns context manager.

## Bases

- `abctools.ContextManager`
- `abctools.AbjadObject`
- `__builtin__.object`

## Special methods

`RedirectedStreams.__enter__()`

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`RedirectedStreams.__exit__(exc_type, exc_value, traceback)`

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

## 7.2 Functions

### 7.2.1 `iotools.clear_terminal`

`iotools.clear_terminal()`  
Runs `clear` if OS is POSIX-compliant (UNIX / Linux / MacOS).  
Runs `cls` if OS is not POSIX-compliant (Windows).

```
>>> iotools.clear_terminal()
```

Returns none.

### 7.2.2 `iotools.count_function_calls`

`iotools.count_function_calls(expr, global_context=None, local_context=None, fixed_point=True)`  
Counts function calls returned by `iotools.profile_expr(expr)`.

**Example 1.** Function calls required to initialize note from string:

```
>>> iotools.count_function_calls("Note('c4')", globals())
10463
```

**Example 2.** Function calls required to initialize note from integers:

```
>>> iotools.count_function_calls("Note(-12, (1, 4))", globals())
158
```

Returns nonnegative integer.

### 7.2.3 iotools.f

`iotools.f(expr)`

Formats *expr* and prints to standard out.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



Returns none.

### 7.2.4 iotools.get\_last\_output\_file\_name

`iotools.get_last_output_file_name(output_directory_path=None)`

Gets last output file name in output directory.

```
>>> iotools.get_last_output_file_name()
'6222.ly'
```

Gets last output file name in Abjad output directory when *output\_directory\_path* is none.

Returns none when output directory contains no output files.

Returns string or none.

### 7.2.5 iotools.get\_next\_output\_file\_name

`iotools.get_next_output_file_name(file_extension='ly', output_directory_path=None)`

Gets next output file name in output directory.

```
>>> iotools.get_next_output_file_name()
'6223.ly'
```

Gets next output file name in Abjad output directory when *output\_directory\_path* is none.

Returns string.

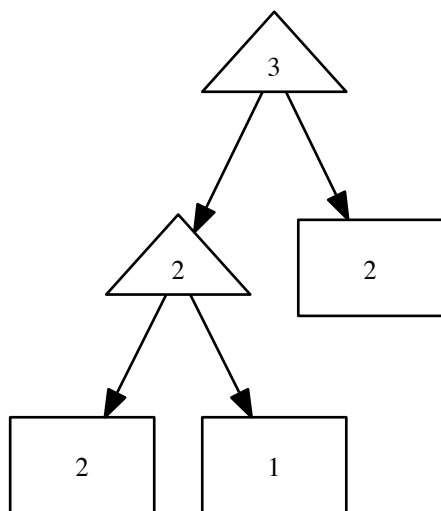
### 7.2.6 iotools.graph

`iotools.graph(expr, image_format='pdf', layout='dot')`

Graphs *expr* with graphviz and opens resulting image in the default image viewer.

```
>>> rtm_syntax = '(3 ((2 (2 1)) 2))'
>>> rhythm_tree = rhythmtreetools.RhythmTreeParser()(rtm_syntax)[0]
>>> print rhythm_tree.pretty_rtm_format
(3 (
  (2 (
    2
    1))
  2))

>>> iotools.graph(rhythm_tree)
```



Returns none.

### 7.2.7 `iotools.insert_expr_into_lilypond_file`

`iotools.insert_expr_into_lilypond_file` (*expr*, *tagline=False*)

Inserts *expr* into LilyPond file.

Returns LilyPond file.

### 7.2.8 `iotools.log`

`iotools.log` ()

Opens the LilyPond log file in operating system-specific text editor.

```
>>> iotools.log()
```

```
GNU LilyPond 2.12.2
Processing `0440.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `0440.ps'...
Converting to `./0440.pdf'...
```

Returns none.

### 7.2.9 `iotools.log_render_lilypond_input`

`iotools.log_render_lilypond_input` (*expr*, *output\_directory\_path=None*, *output\_file\_name\_root=None*, *tagline=False*, *docs=False*)

Writes both .ly and .pdf files to *output\_directory*.

Writes to Abjad output directory when *output\_directory* is none.

Writes to next 4-digit numeric file name when *output\_file\_name\_root* is none.

Returns file name, Abjad format time (in seconds) and LilyPond format time (in seconds).



## 7.2.10 iotools.ly

`iotools.ly` (*target=-1*)

Opens the last LilyPond output file in text editor.

**Example 1.** Open the last LilyPond output file:

```
>>> iotools.ly()

% Abjad revision 2162
% 2009-05-31 14:29

\version "2.12.2"
\include "english.ly"

{
    c'4
}
```

**Example 2.** Open the next-to-last LilyPond output file:

```
>>> iotools.ly(-2)
```

Returns none.

## 7.2.11 iotools.open\_file

`iotools.open_file` (*file\_path*, *application=None*)

Opens *file\_path* with operating system-specific file-opener with *application* is none.

Opens *file\_path* with *application* when *application* is not none.

Returns none.

## 7.2.12 iotools.p

`iotools.p` (*arg*, *language='english'*)

Parses *arg* as LilyPond string.

```
>>> p("{c'4 d'4 e'4 f'4}")
{c'4, d'4, e'4, f'4}
```

```
>>> container = _
```

```
>>> f(container)
{
    c'4
    d'4
    e'4
    f'4
}
```

A pitch-name language may also be specified.

```
>>> p("{c'8 des' e' fis' }", language='nederlands')
{c'8, df'8, e'8, fs'8}
```

Returns Abjad expression.

## 7.2.13 iotools.pdf

`iotools.pdf` (*target=-1*)

Opens the last PDF generated by Abjad with `iotools.pdf()`.

Opens the next-to-last PDF generated by Abjad with `iotools.pdf(-2)`.

Returns `none`.

Abjad writes PDFs to the `~/ .abjad/output` directory by default.

You may change this by setting the `abjad_output` variable in the `config.py` file.

## 7.2.14 iotools.play

`iotools.play(expr)`  
Plays *expr*.

```
>>> note = Note("c'4")
```

```
>>> iotools.play(note)
```

This input creates and opens a one-note MIDI file.

Abjad outputs MIDI files of the format `filename.mid` under Windows.

Abjad outputs MIDI files of the format `filename.midi` under other operating systems.

Returns `none`.

## 7.2.15 iotools.plot

`iotools.plot(expr, image_format='png', width=640, height=320)`  
Plots *expr* with `gnuplot` and opens resulting image in the default image viewer.

Returns `none`.

## 7.2.16 iotools.profile\_expr

`iotools.profile_expr(expr, sort_by='cum', line_count=12, strip_dirs=True, print_callers=False, print_callees=False, global_context=None, local_context=None, print_to_terminal=True)`

Profiles *expr*.

```
>>> iotools.profile_expr('Staff(notetools.make_repeated_notes(8))')
Tue Apr  5 20:32:40 2011      _tmp_abj_profile

      2852 function calls (2829 primitive calls) in 0.006 CPU seconds

Ordered by: cumulative time
List reduced from 118 to 12 due to restriction <12>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   1      0.000    0.000    0.006    0.006 <string>:1(<module>)
   1      0.000    0.000    0.003    0.003 make_repeated_notes.py:5(
   1      0.001    0.001    0.003    0.003 make_notes.py:12(make_not
   1      0.000    0.000    0.003    0.003 Staff.py:21(__init__)
   1      0.000    0.000    0.003    0.003 Context.py:11(__init__)
   1      0.000    0.000    0.003    0.003 Container.py:23(__init__)
   1      0.000    0.000    0.003    0.003 Container.py:271(_initial
   2      0.000    0.000    0.002    0.001 all_are_logical_voice_con
  52      0.001    0.000    0.002    0.000 component_to_logical_voic
   1      0.000    0.000    0.002    0.002 _construct_unprolated_not
   8      0.000    0.000    0.002    0.000 make_tied_note.py:5(make_
   8      0.000    0.000    0.002    0.000 make_tied_leaf.py:5(make_
```

Wraps the built-in Python `cProfile` module.

Set *expr* to any string of Abjad input.

Set *sort\_by* to `'cum'`, `'time'` or `'calls'`.

Set *line\_count* to any nonnegative integer.

Set *strip\_dirs* to true to strip directory names from output lines.

See the [Python docs](#) for more information on the Python profilers.

Returns none when *print\_to\_terminal* is false.

Returns string when *print\_to\_terminal* is true.

### 7.2.17 iotools.redo

`iotools.redo(target=-1, lily_time=10)`

Rerenders the last `.ly` file created in Abjad and then shows the resulting PDF.

**Example 1.** Redo the last LilyPond file created in Abjad:

```
>>> iotools.redo()
```

**Example 2.** Redo the next-to-last LilyPond file created in Abjad:

```
>>> iotools.redo(-2)
```

Returns none.

### 7.2.18 iotools.run\_abjad

`iotools.run_abjad()`

Runs Abjad.

Returns none.

### 7.2.19 iotools.run\_lilypond

`iotools.run_lilypond(lilypond_file_name, lilypond_path)`

Runs LilyPond.

Returns none.

### 7.2.20 iotools.save\_last\_ly\_as

`iotools.save_last_ly_as(file_path)`

Saves last LilyPond file created in Abjad as *file\_path*.

```
>>> file_path = '/project/output/example-1.ly'
>>> iotools.save_last_ly_as(file_path)
```

Returns none.

### 7.2.21 iotools.save\_last\_pdf\_as

`iotools.save_last_pdf_as(file_path)`

Saves last PDF created in Abjad as *file\_path*.

```
>>> file_path = '/project/output/example-1.pdf'
>>> iotools.save_last_pdf_as(file_path)
```

Returns none.

### 7.2.22 iotools.show

`iotools.show` (*expr*, *return\_timing=False*, *suppress\_pdf=False*, *docs=False*)  
Shows *expr*.

**Example 1.** Show a note:

```
>>> note = Note("c'4")
>>> show(note)
```



**Example 2.** Show a note and return Abjad and LilyPond processing times in seconds:

```
>>> staff = Staff(Note("c'4") * 200)
>>> show(note, return_timing=True)
(0, 3)
```



Wraps *expr* in a LilyPond file with settings and overrides suitable for the Abjad reference manual. When *docs* is true.

Abjad writes LilyPond input files to the `~/ .abjad/output` directory by default.

You may change this by setting the `abjad_output` variable in the `config.py` file.

Returns none or timing tuple.

### 7.2.23 iotools.spawn\_subprocess

`iotools.spawn_subprocess` (*command*)  
Spawns subprocess and runs *command*.

Redirects stderr to stdout.

```
>>> iotools.spawn_subprocess('echo "hello world"')
hello world
```

The function is basically a reimplementation of the deprecated `os.system()` using Python's `subprocess` module.

Returns integer result code.

### 7.2.24 iotools.verify\_output\_directory

`iotools.verify_output_directory` (*directory*)  
Verifies output *directory*.

Returns none.

### 7.2.25 iotools.warn\_almost\_full

`iotools.warn_almost_full` (*last\_number*)  
Prints warning when Abjad output directory is almost full.

Returns none.

### 7.2.26 iotools.which

`iotools.which(name, flags=1)`

Finds executable *name*.

Similar to Unix `which` command.

```
>>> iotools.which('python2.7')
['/usr/bin/python2.7']
```

Returns list of zero or more full paths to *name*.

### 7.2.27 iotools.write\_expr\_to\_ly

`iotools.write_expr_to_ly(expr, file_name, print_status=False, tagline=False, docs=False)`

Writes *expr* to *file\_name*.

```
>>> note = Note("c'4")
>>> iotools.write_expr_to_ly(note, '/home/user/foo.ly')
```

Returns `none`.

### 7.2.28 iotools.write\_expr\_to\_pdf

`iotools.write_expr_to_pdf(expr, file_name, print_status=False, tagline=False)`

Writes *expr* to PDF *file\_name*.

```
>>> note = Note("c'4")
>>> iotools.write_expr_to_pdf(note, 'one_note.pdf')
```

Returns `none`.

### 7.2.29 iotools.z

`iotools.z(expr)`

Prints the storage format of *expr*.

Returns `none`.



# ITERATIONTOOLS

## 8.1 Functions

### 8.1.1 iterationtools.iterate\_chords\_in\_expr

`iterationtools.iterate_chords_in_expr` (*expr*, *reverse=False*, *start=0*, *stop=None*)

Iterate chords forward in *expr*:

```
>>> staff = Staff("<e' g' c''>8 a'8 r8 <d' f' b'>8 r2")
```

```
>>> for chord in iterationtools.iterate_chords_in_expr(staff):
...     chord
Chord("<e' g' c''>8")
Chord("<d' f' b'>8")
```

Iterate chords backward in *expr*:

```
::
```

```
>>> for chord in iterationtools.iterate_chords_in_expr(staff, reverse=True):
...     chord
Chord("<d' f' b'>8")
Chord("<e' g' c''>8")
```

Iterates across different logical voices.

Returns generator.

### 8.1.2 iterationtools.iterate\_components\_and\_grace\_containers\_in\_expr

`iterationtools.iterate_components_and_grace_containers_in_expr` (*expr*,  
*component\_class*)

Iterate components of *component\_class* forward in *expr*:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spannertools.BeamSpanner(voice[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> grace_notes = [Note("c'16"), Note("d'16")]
>>> containertools.GraceContainer(grace_notes, kind='grace')(voice[1])
Note("d'8")
```

```
>>> after_grace_notes = [Note("e'16"), Note("f'16")]
>>> containertools.GraceContainer(after_grace_notes, kind='after')(voice[1])
Note("d'8")
```

```
>>> x = iterationtools.iterate_components_and_grace_containers_in_expr(voice, Note)
>>> for note in x:
...     note
...
Note("c'8")
Note("c'16")
Note("d'16")
Note("d'8")
Note("e'16")
Note("f'16")
Note("e'8")
Note("f'8")
```

Include grace leaves before main leaves.

Include grace leaves after main leaves.

### 8.1.3 iterationtools.iterate\_components\_depth\_first

`iterationtools.iterate_components_depth_first` (*component*, *capped=True*,  
*unique=True*, *forbid=None*, *direction=Left*)

Iterate components depth-first from *component*.

---

#### Todo

Add usage examples.

---

### 8.1.4 iterationtools.iterate\_components\_in\_expr

`iterationtools.iterate_components_in_expr` (*expr*, *component\_class=None*, *reverse=False*, *start=0*, *stop=None*)

Iterate components forward in *expr*.

### 8.1.5 iterationtools.iterate\_containers\_in\_expr

`iterationtools.iterate_containers_in_expr` (*expr*, *reverse=False*, *start=0*, *stop=None*)

Iterate containers forward in *expr*:

```
>>> staff = Staff([Voice("c'8 d'8"), Voice("e'8 f'8 g'8")])
>>> Tuplet(Fraction(2, 3), staff[1][:])
Tuplet(2/3, [e'8, f'8, g'8])
>>> staff.is_simultaneous = True
```

```
>>> for x in iterationtools.iterate_containers_in_expr(staff):
...     x
Staff<<2>>
Voice{2}
Voice{1}
Tuplet(2/3, [e'8, f'8, g'8])
```

Iterate containers backward in *expr*:

```
>>> for x in iterationtools.iterate_containers_in_expr(staff, reverse=True):
...     x
Staff<<2>>
Voice{1}
Tuplet(2/3, [e'8, f'8, g'8])
Voice{2}
```

Iterates across different logical voices.

Returns generator.



### 8.1.6 iterationtools.iterate\_contexts\_in\_expr

`iterationtools.iterate_contexts_in_expr(expr, reverse=False, start=0, stop=None)`

Iterate contexts forward in *expr*:

```
>>> staff = Staff([Voice("c'8 d'8"), Voice("e'8 f'8 g'8")])
>>> Tuplet(Fraction(2, 3), staff[1][:])
Tuplet(2/3, [e'8, f'8, g'8])
>>> staff.is_simultaneous = True
```

```
>>> for x in iterationtools.iterate_contexts_in_expr(staff):
...     x
Staff<<2>>
Voice{2}
Voice{1}
```

Iterate contexts backward in *expr*:

```
>>> for x in iterationtools.iterate_contexts_in_expr(staff, reverse=True):
...     x
Staff<<2>>
Voice{1}
Voice{2}
```

Iterates across different logical voices.

Returns generator.

### 8.1.7 iterationtools.iterate\_leaf\_pairs\_in\_expr

`iterationtools.iterate_leaf_pairs_in_expr(expr)`

Iterate leaf pairs forward in *expr*:

```
>>> score = Score([])
>>> notes = [Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"), Note("g'4")]
>>> score.append(Staff(notes))
>>> notes = [Note(x, (1, 4)) for x in [-12, -15, -17]]
>>> score.append(Staff(notes))
>>> contexttools.ClefMark('bass')(score[1])
ClefMark('bass')(Staff{3})
```

```
>>> for pair in iterationtools.iterate_leaf_pairs_in_expr(score):
...     pair
(Note("c'8"), Note('c4'))
(Note("c'8"), Note("d'8"))
(Note('c4'), Note("d'8"))
(Note("d'8"), Note("e'8"))
(Note("d'8"), Note('a,4'))
(Note('c4'), Note("e'8"))
(Note('c4'), Note('a,4'))
(Note("e'8"), Note('a,4'))
(Note("e'8"), Note("f'8"))
(Note('a,4'), Note("f'8"))
(Note("f'8"), Note("g'4"))
(Note("f'8"), Note('g,4'))
(Note('a,4'), Note("g'4"))
(Note('a,4'), Note('g,4'))
(Note("g'4"), Note('g,4'))
```

Iterate leaf pairs left-to-right and top-to-bottom.

Returns generator.

### 8.1.8 iterationtools.iterate\_leaves\_in\_expr

`iterationtools.iterate_leaves_in_expr(expr, reverse=False, start=0, stop=None)`

Iterate leaves forward in *expr*:

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 || 2/8 g'8 a'8 |")
>>> f(staff)
\new Staff {
  {
    \time 2/8
    c'8
    d'8
  }
  {
    e'8
    f'8
  }
  {
    g'8
    a'8
  }
}
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff):
...     leaf
...
Note("c'8")
Note("d'8")
Note("e'8")
Note("f'8")
Note("g'8")
Note("a'8")
```

Use the optional *start* and *stop* keyword parameters to control the start and stop indices of iteration.

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff, start=3):
...     leaf
...
Note("f'8")
Note("g'8")
Note("a'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff, start=0, stop=3):
...     leaf
...
Note("c'8")
Note("d'8")
Note("e'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff, start=2, stop=4):
...     leaf
...
Note("e'8")
Note("f'8")
```

Iterate leaves backward in *expr*:

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff, reverse=True):
...     leaf
...
Note("a'8")
Note("g'8")
Note("f'8")
Note("e'8")
Note("d'8")
Note("c'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(
...     staff, start=3, reverse=True):
...     leaf
...
Note("e'8")
Note("d'8")
Note("c'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(
...     staff, start=0, stop=3, reverse=True):
...     leaf
...
Note("a'8")
Note("g'8")
Note("f'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(
...     staff, start=2, stop=4, reverse=True):
...     leaf
...
Note("f'8")
Note("e'8")
```

Iterates across different logical voices.

Returns generator.

### 8.1.9 iterationtools.iterate\_logical\_voice\_from\_component

`iterationtools.iterate_logical_voice_from_component` (*component*, *component\_class=None*, *reverse=False*)

Iterate logical voice forward from *component* and yield instances of *component\_class*.

```
>>> container_1 = Container([Voice("c'8 d'8"), Voice("e'8 f'8")])
>>> container_1.is_simultaneous = True
>>> container_1[0].name = 'voice 1'
>>> container_1[1].name = 'voice 2'
>>> container_2 = Container([Voice("g'8 a'8"), Voice("b'8 c'8")])
>>> container_2.is_simultaneous = True
>>> container_2[0].name = 'voice 1'
>>> container_2[1].name = 'voice 2'
>>> staff = Staff([container_1, container_2])
>>> show(staff)
```



Starting from the first leaf in score:

```
>>> for x in iterationtools.iterate_logical_voice_from_component(
...     staff.select_leaves(allow_discontiguous_leaves=True)[0], Note):
...     x
...
Note("c'8")
Note("d'8")
Note("g'8")
Note("a'8")
```

Starting from the second leaf in score:

```
>>> for x in iterationtools.iterate_logical_voice_from_component(
...     staff.select_leaves(allow_discontiguous_leaves=True)[1], Note):
...     x
...
Note("d'8")
Note("g'8")
Note("a'8")
```

Yield all components in logical voice:

```
>>> for x in iterationtools.iterate_logical_voice_from_component(
...     staff.select_leaves(allow_discontiguous_leaves=True)[0]):
...     x
...
Note("c'8")
Voice="voice 1"{2}
```

```
Note("d'8")
Voice="voice 1"{2}
Note("g'8")
Note("a'8")
```

Iterate logical voice backward from *component* and yield instances of *component\_class*, starting from the last leaf in score:

```
>>> for x in iterationtools.iterate_logical_voice_from_component(
...     staff.select_leaves(allow_discontiguous_leaves=True)[-1],
...     Note,
...     reverse=True,
...     ):
...     x
Note("c'8")
Note("b'8")
Note("f'8")
Note("e'8")
```

Yield all components in logical voice:

```
>>> for x in iterationtools.iterate_logical_voice_from_component(
...     staff.select_leaves(allow_discontiguous_leaves=True)[-1],
...     reverse=True,
...     ):
...     x
Note("c'8")
Voice="voice 2"{2}
Note("b'8")
Voice="voice 2"{2}
Note("f'8")
Note("e'8")
```

Returns generator.

### 8.1.10 iterationtools.iterate\_logical\_voice\_in\_expr

`iterationtools.iterate_logical_voice_in_expr(expr, component_class, logical_voice_indicator, reverse=False)`  
Yield left-to-right instances of *component\_class* in *expr* with *logical\_voice\_indicator*:

```
>>> container_1 = Container([Voice("c'8 d'8"), Voice("e'8 f'8")])
>>> container_1.is_simultaneous = True
>>> container_1[0].name = 'voice 1'
>>> container_1[1].name = 'voice 2'
>>> container_2 = Container([Voice("g'8 a'8"), Voice("b'8 c'8")])
>>> container_2.is_simultaneous = True
>>> container_2[0].name = 'voice 1'
>>> container_2[1].name = 'voice 2'
>>> staff = Staff([container_1, container_2])
>>> show(staff)
```



```
>>> leaf = staff.select_leaves(allow_discontiguous_leaves=True)[0]
>>> signature = inspect(leaf).get_parentage().logical_voice_indicator
>>> for x in iterationtools.iterate_logical_voice_in_expr(
...     staff, Note, signature):
...     x
...
Note("c'8")
Note("d'8")
Note("g'8")
Note("a'8")
```

Returns generator.

### 8.1.11 iterationtools.iterate\_measures\_in\_expr

`iterationtools.iterate_measures_in_expr` (*expr*, *reverse=False*, *start=0*, *stop=None*)

Iterate measures forward in *expr*:

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 || 2/8 g'8 a'8 |")
```

```
>>> for measure in iterationtools.iterate_measures_in_expr(staff):
...     measure
...
Measure(2/8, [c'8, d'8])
Measure(2/8, [e'8, f'8])
Measure(2/8, [g'8, a'8])
```

Use the optional *start* and *stop* keyword parameters to control the start and stop indices of iteration.

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, start=1):
...     measure
...
Measure(2/8, [e'8, f'8])
Measure(2/8, [g'8, a'8])
```

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, start=0, stop=2):
...     measure
...
Measure(2/8, [c'8, d'8])
Measure(2/8, [e'8, f'8])
```

Iterate measures backward in *expr*:

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, reverse=True):
...     measure
...
Measure(2/8, [g'8, a'8])
Measure(2/8, [e'8, f'8])
Measure(2/8, [c'8, d'8])
```

Use the optional *start* and *stop* keyword parameters to control indices of iteration.

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, start=1, reverse=True):
...     measure
...
Measure(2/8, [e'8, f'8])
Measure(2/8, [c'8, d'8])
```

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, start=0, stop=2, reverse=True):
...     measure
...
Measure(2/8, [g'8, a'8])
Measure(2/8, [e'8, f'8])
```

Iterates across different logical voices.

Returns generator.

### 8.1.12 iterationtools.iterate\_nontrivial\_tie\_chains\_in\_expr

`iterationtools.iterate_nontrivial_tie_chains_in_expr` (*expr*, *reverse=False*)

Iterate nontrivial tie chains forward in *expr*:

```
>>> staff = Staff(r"c'4 ~ \times 2/3 { c'16 d'8 } e'8 f'4 ~ f'16")
```

```
>>> for x in \
...     iterationtools.iterate_nontrivial_tie_chains_in_expr(staff):
...     x
...
TieChain(Note("c'4"), Note("c'16"))
TieChain(Note("f'4"), Note("f'16"))
```

Iterate nontrivial tie chains backward in *expr*:

```
>>> for x in \
...     iterationtools.iterate_nontrivial_tie_chains_in_expr(
...         staff, reverse=True):
...     x
...
TieChain(Note("f'4"), Note("f'16"))
TieChain(Note("c'4"), Note("c'16"))
```

Returns generator.

### 8.1.13 iterationtools.iterate\_notes\_and\_chords\_in\_expr

`iterationtools.iterate_notes_and_chords_in_expr(expr, reverse=False, start=0, stop=None)`

Iterate notes and chords forward in *expr*:

```
>>> staff = Staff("<e' g' c''>8 a'8 r8 <d' f' b'>8 r2")
```

```
>>> for leaf in iterationtools.iterate_notes_and_chords_in_expr(staff):
...     leaf
Chord("<e' g' c''>8")
Note("a'8")
Chord("<d' f' b'>8")
```

Iterate notes and chords backward in *expr*:

```
>>> for leaf in iterationtools.iterate_notes_and_chords_in_expr(staff, reverse=True):
...     leaf
Chord("<d' f' b'>8")
Note("a'8")
Chord("<e' g' c''>8")
```

Iterates across different logical voices.

Returns generator.

### 8.1.14 iterationtools.iterate\_notes\_in\_expr

`iterationtools.iterate_notes_in_expr(expr, reverse=False, start=0, stop=None)`

Yield left-to-right notes in *expr*:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 8), "c'8 d'8"))
>>> staff.append(Measure((2, 8), "e'8 f'8"))
>>> staff.append(Measure((2, 8), "g'8 a'8"))
```

```
>>> for note in iterationtools.iterate_notes_in_expr(staff):
...     note
...
Note("c'8")
Note("d'8")
Note("e'8")
Note("f'8")
Note("g'8")
Note("a'8")
```

Use optional *start* and *stop* keyword parameters to control start and stop indices of iteration:

```
>>> for note in iterationtools.iterate_notes_in_expr(staff, start=3):
...     note
...
Note("f'8")
Note("g'8")
Note("a'8")
```

```
>>> for note in iterationtools.iterate_notes_in_expr(
...     staff, start=0, stop=3):
...     note
...
Note("c'8")
Note("d'8")
Note("e'8")
```

```
>>> for note in iterationtools.iterate_notes_in_expr(
...     staff, start=2, stop=4):
...     note
...
Note("e'8")
Note("f'8")
```

Yield right-to-left notes in *expr*:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 8), "c'8 d'8"))
>>> staff.append(Measure((2, 8), "e'8 f'8"))
>>> staff.append(Measure((2, 8), "g'8 a'8"))
```

```
>>> for note in iterationtools.iterate_notes_in_expr(
...     staff, reverse=True):
...     note
...
Note("a'8")
Note("g'8")
Note("f'8")
Note("e'8")
Note("d'8")
Note("c'8")
```

Use optional *start* and *stop* keyword parameters to control indices of iteration:

```
>>> for note in iterationtools.iterate_notes_in_expr(
...     staff, reverse=True, start=3):
...     note
...
Note("e'8")
Note("d'8")
Note("c'8")
```

```
>>> for note in iterationtools.iterate_notes_in_expr(
...     staff, reverse=True, start=0, stop=3):
...     note
...
Note("a'8")
Note("g'8")
Note("f'8")
```

```
>>> for note in iterationtools.iterate_notes_in_expr(
...     staff, reverse=True, start=2, stop=4):
...     note
...
Note("f'8")
Note("e'8")
```

Iterates across different logical voices.

Returns generator.

### 8.1.15 `iterationtools.iterate_pitched_tie_chains_in_expr`

`iterationtools.iterate_pitched_tie_chains_in_expr` (*expr*, *reverse=False*)

Iterate pitched tie chains forward in *expr*:

```
>>> staff = Staff(r"c'4 ~ \times 2/3 { c'16 d'8 }")
>>> staff.extend(r"e'8 r8 f'8 ~ f'16 r8.")
```

```
>>> for x in iterationtools.iterate_pitched_tie_chains_in_expr(staff):
...     x
...
TieChain(Note("c'4"), Note("c'16"))
TieChain(Note("d'8"),)
TieChain(Note("e'8"),)
TieChain(Note("f'8"), Note("f'16"))
```

Iterate pitched tie chains backward in *expr*:

```
>>> for x in iterationtools.iterate_pitched_tie_chains_in_expr(
...     staff, reverse=True):
...     x
...
TieChain(Note("f'8"), Note("f'16"))
TieChain(Note("e'8"),)
TieChain(Note("d'8"),)
TieChain(Note("c'4"), Note("c'16"))
```

Tie chains are pitched if they comprise notes or chords.

Tie chains are not pitched if they comprise rests or skips.

Returns generator.

### 8.1.16 `iterationtools.iterate_rests_in_expr`

`iterationtools.iterate_rests_in_expr` (*expr*, *reverse=False*, *start=0*, *stop=None*)

Iterate rests forward in *expr*:

```
>>> staff = Staff("<e' g' c''>8 a'8 r8 <d' f' b'>8 r2")
```

```
>>> for rest in iterationtools.iterate_rests_in_expr(staff):
...     rest
Rest('r8')
Rest('r2')
```

Iterate rests backward in *expr*:

```
>>> for rest in iterationtools.iterate_rests_in_expr(
...     staff, reverse=True):
...     rest
Rest('r2')
Rest('r8')
```

Iterates across different logical voices.

Returns generator.

### 8.1.17 `iterationtools.iterate_runs_in_expr`

`iterationtools.iterate_runs_in_expr` (*sequence*, *classes*)

Iterate runs in expression.

**Example 1.** Iterate runs of notes and chords at only the top level of score:

```
>>> staff = Staff(r"\times 2/3 { c'8 d'8 r8 }")
>>> staff.append(r"\times 2/3 { r8 <e' g'>8 <f' a'>8 }")
>>> staff.extend("g'8 a'8 r8 r8 <b' d''>8 <c'' e''>8")
```



```
>>> for group in iterationtools.iterate_runs_in_expr(
...     staff[:], (Note, Chord)):
...     group
(Note("g'8"), Note("a'8"))
(Chord("<b' d''>8"), Chord("<c' e''>8"))
```

**Example 2.** Iterate runs of notes and chords at all levels of score:

```
>>> leaves = iterationtools.iterate_leaves_in_expr(staff)
```

```
>>> for group in iterationtools.iterate_runs_in_expr(
...     leaves, (Note, Chord)):
...     group
(Note("c'8"), Note("d'8"))
(Chord("<e' g'>8"), Chord("<f' a'>8"), Note("g'8"), Note("a'8"))
(Chord("<b' d''>8"), Chord("<c' e''>8"))
```

Returns generator.

### 8.1.18 iterationtools.iterate\_scores\_in\_expr

`iterationtools.iterate_scores_in_expr(expr, reverse=False, start=0, stop=None)`

Iterate scores forward in *expr*:

```
>>> score_1 = Score([Staff("c'8 d'8 e'8 f'8")])
>>> score_2 = Score([Staff("c'1"), Staff("g'1")])
>>> scores = [score_1, score_2]
```

```
>>> for score in iterationtools.iterate_scores_in_expr(scores):
...     score
Score<<1>>
Score<<2>>
```

Iterate scores backward in *expr*:

```
::
```

```
>>> for score in iterationtools.iterate_scores_in_expr(scores, reverse=True):
...     score
Score<<2>>
Score<<1>>
```

Iterates across different logical voices.

Returns generator.

### 8.1.19 iterationtools.iterate\_semantic\_voices\_in\_expr

`iterationtools.iterate_semantic_voices_in_expr(expr, reverse=False, start=0, stop=None)`

Iterate semantic voices forward in *expr*:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(3, 8), (5, 16), (5, 16)])
>>> time_signature_voice = Voice(measures)
>>> time_signature_voice.name = 'TimeSignatureVoice'
>>> time_signature_voice.is_nonsemantic = True
>>> music_voice = Voice("c'4. d'4 e'16 f'4 g'16")
>>> music_voice.name = 'MusicVoice'
>>> staff = Staff([time_signature_voice, music_voice])
>>> staff.is_simultaneous = True
```

Iterate semantic voices backward in *expr*:

```
>>> for voice in iterationtools.iterate_semantic_voices_in_expr(staff, reverse=True):
...     voice
Voice-"MusicVoice"{5}
```

Returns generator.

### 8.1.20 `iterationtools.iterate_skips_in_expr`

`iterationtools.iterate_skips_in_expr` (*expr*, *reverse=False*, *start=0*, *stop=None*)  
Iterate skips forward in *expr*:

```
>>> staff = Staff("<e' g' c''>8 a'8 s8 <d' f' b'>8 s2")
```

```
>>> for skip in iterationtools.iterate_skips_in_expr(staff):  
...     skip  
Skip('s8')  
Skip('s2')
```

Iterate skips backwards in *expr*:

```
>>> for skip in iterationtools.iterate_skips_in_expr(staff, reverse=True):  
...     skip  
Skip('s2')  
Skip('s8')
```

Iterates across different logical voices.

Returns generator.

### 8.1.21 `iterationtools.iterate_staves_in_expr`

`iterationtools.iterate_staves_in_expr` (*expr*, *reverse=False*, *start=0*, *stop=None*)  
Iterate staves forward in *expr*:

```
>>> score = Score(4 * Staff([]))
```

```
>>> for staff in iterationtools.iterate_staves_in_expr(score):  
...     staff  
...  
Staff{}  
Staff{}  
Staff{}  
Staff{}
```

Iterate staves backward in *expr*:

```
::
```

```
>>> for staff in iterationtools.iterate_staves_in_expr(score, reverse=True):  
...     staff  
...  
Staff{}  
Staff{}  
Staff{}  
Staff{}
```

Returns generator.

### 8.1.22 `iterationtools.iterate_tie_chains_in_expr`

`iterationtools.iterate_tie_chains_in_expr` (*expr*, *reverse=False*)  
Iterate tie chains forward in *expr*:

```
>>> staff = Staff(r"c'4 ~ \times 2/3 { c'16 d'8 } e'8 f'4 ~ f'16")
```

```
>>> for x in iterationtools.iterate_tie_chains_in_expr(staff):
...     x
...
TieChain(Note("c'4"), Note("c'16"))
TieChain(Note("d'8"),)
TieChain(Note("e'8"),)
TieChain(Note("f'4"), Note("f'16"))
```

Iterate tie chains backward in *expr*:

```
>>> for x in iterationtools.iterate_tie_chains_in_expr(staff, reverse=True):
...     x
...
TieChain(Note("f'4"), Note("f'16"))
TieChain(Note("e'8"),)
TieChain(Note("d'8"),)
TieChain(Note("c'4"), Note("c'16"))
```

Returns generator.

### 8.1.23 iterationtools.iterate\_timeline\_from\_component

`iterationtools.iterate_timeline_from_component` (*expr*, *component\_class=None*, *reverse=False*)

Iterate timeline forward from *component*:

```
>>> score = Score([])
>>> score.append(Staff("c'4 d'4 e'4 f'4"))
>>> score.append(Staff("g'8 a'8 b'8 c'8"))
```

```
>>> for leaf in iterationtools.iterate_timeline_from_component(
...     score[1][2]):
...     leaf
...
Note("b'8")
Note("c'8")
Note("e'4")
Note("f'4")
```

Iterate timeline backward from *component*:

```
::
```

```
>>> for leaf in iterationtools.iterate_timeline_from_component(
...     score[1][2], reverse=True):
...     leaf
...
Note("b'8")
Note("c'4")
Note("a'8")
Note("g'8")
```

Yield components sorted backward by score offset stop time when *reverse* is True.

Iterate leaves when *component\_class* is none.

---

#### Todo

optimize to avoid behind-the-scenes full-score traversal.

---

### 8.1.24 iterationtools.iterate\_timeline\_in\_expr

`iterationtools.iterate_timeline_in_expr` (*expr*, *component\_class=None*, *reverse=False*)

Iterate timeline forward in *expr*:

```
>>> score = Score([])
>>> score.append(Staff("c'4 d'4 e'4 f'4"))
>>> score.append(Staff("g'8 a'8 b'8 c'8"))
```

```
>>> for leaf in iterationtools.iterate_timeline_in_expr(score):
...     leaf
...
Note("c'4")
Note("g'8")
Note("a'8")
Note("d'4")
Note("b'8")
Note("c'8")
Note("e'4")
Note("f'4")
```

Iterate timeline backward in *expr*:

```
::
```

```
>>> for leaf in iterationtools.iterate_timeline_in_expr(
...     score, reverse=True):
...     leaf
...
Note("f'4")
Note("e'4")
Note("d'4")
Note("c'8")
Note("b'8")
Note("c'4")
Note("a'8")
Note("g'8")
```

Iterate leaves when *component\_class* is none.

---

### Todo

optimize to avoid behind-the-scenes full-score traversal.

---

## 8.1.25 iterationtools.iterate\_topmost\_tie\_chains\_and\_components\_in\_expr

`iterationtools.iterate_topmost_tie_chains_and_components_in_expr(expr)`

Iterate topmost tie chains and components forward in *expr*:

```
>>> string = r"c'8 ~ c'32 d'8 ~ d'32 \times 2/3 { e'8 f'8 g'8 } a'8 ~ a'32 b'8 ~ b'32"
>>> staff = Staff(string)
```

```
>>> for x in \
...     iterationtools.iterate_topmost_tie_chains_and_components_in_expr(
...         staff):
...     x
...
TieChain(Note("c'8"), Note("c'32"))
TieChain(Note("d'8"), Note("d'32"))
Tuplet(2/3, [e'8, f'8, g'8])
TieChain(Note("a'8"), Note("a'32"))
TieChain(Note("b'8"), Note("b'32"))
```

Raise tie chain error on overlapping tie chains.

Returns generator.

### 8.1.26 iterationtools.iterate\_tuplets\_in\_expr

`iterationtools.iterate_tuplets_in_expr` (*expr*, *reverse=False*, *start=0*, *stop=None*)

Iterate tuplets forward in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> Tuplet(Fraction(2, 3), staff[:3])
Tuplet(2/3, [c'8, d'8, e'8])
>>> Tuplet(Fraction(2, 3), staff[-3:])
Tuplet(2/3, [a'8, b'8, c''8])

>>> f(staff)
\new Staff {
  \times 2/3 {
    c'8
    d'8
    e'8
  }
  f'8
  g'8
  \times 2/3 {
    a'8
    b'8
    c''8
  }
}

>>> for tuplet in iterationtools.iterate_tuplets_in_expr(staff):
...     tuplet
...
Tuplet(2/3, [c'8, d'8, e'8])
Tuplet(2/3, [a'8, b'8, c''8])
```

Iterate tuplets backward in *expr*:

```
>>> for tuplet in iterationtools.iterate_tuplets_in_expr(
...     staff, reverse=True):
...     tuplet
...
Tuplet(2/3, [a'8, b'8, c''8])
Tuplet(2/3, [c'8, d'8, e'8])
```

Iterates across different logical voices.

Returns generator.

### 8.1.27 iterationtools.iterate\_vertical\_moments\_in\_expr

`iterationtools.iterate_vertical_moments_in_expr` (*expr*, *reverse=False*)

Iterate vertical moments forward in *expr*:

```
>>> score = Score([])
>>> staff = Staff(r"\times 4/3 { d'8 c'8 b'8 }")
>>> score.append(staff)

>>> piano_staff = scoretools.PianoStaff([])
>>> piano_staff.append(Staff("a'4 g'4"))
>>> piano_staff.append(Staff(r"""\clef "bass" f'8 e'8 d'8 c'8"""))
>>> score.append(piano_staff)

>>> for x in iterationtools.iterate_vertical_moments_in_expr(score):
...     x.leaves
...
(Note("d'8"), Note("a'4"), Note("f'8"))
(Note("d'8"), Note("a'4"), Note("e'8"))
(Note("c'8"), Note("a'4"), Note("e'8"))
(Note("c'8"), Note("g'4"), Note("d'8"))
(Note("b'8"), Note("g'4"), Note("d'8"))
(Note("b'8"), Note("g'4"), Note("c'8"))
```

```
>>> for x in iterationtools.iterate_vertical_moments_in_expr(
...     piano_staff):
...     x.leaves
...
(Note("a'4"), Note("f'8"))
(Note("a'4"), Note("e'8"))
(Note("g'4"), Note("d'8"))
(Note("g'4"), Note("c'8"))
```

Iterate vertical moments backward in *expr*:

```
::
```

```
>>> for x in iterationtools.iterate_vertical_moments_in_expr(
...     score, reverse=True):
...     x.leaves
...
(Note("b'8"), Note("g'4"), Note("c'8"))
(Note("b'8"), Note("g'4"), Note("d'8"))
(Note("c''8"), Note("g'4"), Note("d'8"))
(Note("c''8"), Note("a'4"), Note("e'8"))
(Note("d''8"), Note("a'4"), Note("e'8"))
(Note("d''8"), Note("a'4"), Note("f'8"))
```

```
>>> for x in iterationtools.iterate_vertical_moments_in_expr(
...     piano_staff, reverse=True):
...     x.leaves
...
(Note("g'4"), Note("c'8"))
(Note("g'4"), Note("d'8"))
(Note("a'4"), Note("e'8"))
(Note("a'4"), Note("f'8"))
```

Returns generator.

### 8.1.28 iterationtools.iterate\_voices\_in\_expr

`iterationtools.iterate_voices_in_expr` (*expr*, *reverse=False*, *start=0*, *stop=None*)

Iterate voices forward in *expr*:

```
>>> voice_1 = Voice("c'8 d'8 e'8 f'8")
>>> voice_2 = Voice("c'4 b4")
>>> staff = Staff([voice_1, voice_2])
>>> staff.is_simultaneous = True
```

```
>>> for voice in iterationtools.iterate_voices_in_expr(staff):
...     voice
Voice{4}
Voice{2}
```

Iterate voices backward in *expr*:

```
::
```

```
>>> for voice in iterationtools.iterate_voices_in_expr(
...     staff, reverse=True):
...     voice
Voice{2}
Voice{4}
```

Returns generator.

# LABELTOOLS

## 9.1 Functions

### 9.1.1 `labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map`

`labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map` (*chord*,  
*color\_map*)

Color *chord* note heads by pitch-class *color\_map*:

```
>>> chord = Chord([12, 14, 18, 21, 23], (1, 4))
```

```
>>> pitches = [[-12, -10, 4], [-2, 8, 11, 17], [19, 27, 30, 33, 37]]
>>> colors = ['red', 'blue', 'green']
>>> color_map = pitchtools.NumberedPitchClassColorMap(pitches, colors)
```

```
>>> labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map(chord, color_map)
Chord("<c' d' fs' a' b'>4")
```

```
>>> show(chord)
```



Also works on notes:

```
>>> note = Note("c'4")
```

```
>>> labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map(note, color_map)
Note("c'4")
```

```
>>> show(note)
```



When *chord* is neither a chord nor note return *chord* unchanged:

```
>>> staff = Staff([])
```

```
>>> labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map(staff, color_map)
Staff{}
```

Return *chord*.

### 9.1.2 `labeltools.color_contents_of_container`

`labeltools.color_contents_of_container` (*container*, *color*)

Color contents of *container*:

```
>>> measure = Measure((2, 8), "c'8 d'8")
```

```
>>> labeltools.color_contents_of_container(measure, 'red')
Measure(2/8, [c'8, d'8])
```

```
>>> show(measure)
```



Returns none.

### 9.1.3 labeltools.color\_leaf

`labeltools.color_leaf(leaf, color)`

Color note:

```
>>> note = Note("c'4")
```

```
>>> labeltools.color_leaf(note, 'red')
Note("c'4")
```

```
>>> show(note)
```



Color rest:

```
>>> rest = Rest('r4')
```

```
>>> labeltools.color_leaf(rest, 'red')
Rest('r4')
```

```
>>> show(rest)
```



Color chord:

```
>>> chord = Chord("<c' e' bf'>4")
```

```
>>> labeltools.color_leaf(chord, 'red')
Chord("<c' e' bf'>4")
```

```
>>> show(chord)
```



Return *leaf*.

### 9.1.4 labeltools.color\_leaves\_in\_expr

`labeltools.color_leaves_in_expr(expr, color)`

Color leaves in *expr*:

```
>>> staff = Staff("cs'8. [ r8. s8. <c' cs' a'>8. ]")
```

```
>>> show(staff)
```





```
>>> labeltools.color_leaves_in_expr(staff, 'red')
```

```
>>> show(staff)
```



Returns none.

### 9.1.5 labeltools.color\_measure

`labeltools.color_measure(measure, color='red')`

Color *measure* with *color*:

```
>>> measure = Measure((2, 8), "c'8 d'8")
```

```
>>> show(measure)
```



```
>>> labeltools.color_measure(measure, 'red')
Measure(2/8, [c'8, d'8])
```

```
>>> show(measure)
```



Returns colored *measure*.

Color names appear in LilyPond Learning Manual appendix B.5.

### 9.1.6 labeltools.color\_measures\_with\_non\_power\_of\_two\_denominators\_in\_expr

`labeltools.color_measures_with_non_power_of_two_denominators_in_expr(expr, color='red')`

Color measures with non-power-of-two denominators in *expr* with *color*:

```
>>> staff = Staff(Measure((2, 8), "c'8 d'8") * 2)
>>> measuretools.scale_measure_denominator_and_adjust_measure_contents(staff[1], 3)
Measure(3/12, [c'8., d'8.])
```

```
>>> show(staff)
```



```
>>> labeltools.color_measures_with_non_power_of_two_denominators_in_expr(staff, 'red')
[Measure(3/12, [c'8., d'8.])]
```

```
>>> show(staff)
```



Returns list of measures colored.

Color names appear in LilyPond Learning Manual appendix B.5.

### 9.1.7 `labeltools.color_note_head_by_numbered_pitch_class_color_map`

`labeltools.color_note_head_by_numbered_pitch_class_color_map` (*pitch\_carrier*)  
Color *pitch\_carrier* note head:

```
>>> note = Note("c'4")
```

```
>>> labeltools.color_note_head_by_numbered_pitch_class_color_map(note)
Note("c'4")
```

```
>>> show(note)
```



Numbered pitch-class color map:

```
0: red
1: MediumBlue
2: orange
3: LightSlateBlue
4: ForestGreen
5: MediumOrchid
6: firebrick
7: DeepPink
8: DarkOrange
9: IndianRed
10: CadetBlue
11: SeaGreen
12: LimeGreen
```

Numbered pitch-class color map can not be changed.

Raise type error when *pitch\_carrier* is not a pitch carrier.

Raise extra pitch error when *pitch\_carrier* carries more than 1 note head.

Raise missing pitch error when *pitch\_carrier* carries no note head.

Return *pitch\_carrier*.

### 9.1.8 `labeltools.label_leaves_in_expr_with_leaf_depth`

`labeltools.label_leaves_in_expr_with_leaf_depth` (*expr*, *markup\_direction=Down*)  
Label leaves in *expr* with leaf depth:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8")
>>> tuplettools.FixedDurationTuplet(Duration(2, 8), staff[-3:])
FixedDurationTuplet(1/4, [e'8, f'8, g'8])
```

```
>>> labeltools.label_leaves_in_expr_with_leaf_depth(staff)
```

```
>>> show(staff)
```



Returns none.

### 9.1.9 `labeltools.label_leaves_in_expr_with_leaf_duration`

`labeltools.label_leaves_in_expr_with_leaf_duration` (*expr*,  
*markup\_direction=Down*)

Label leaves in *expr* with prolated leaf duration:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> labeltools.label_leaves_in_expr_with_leaf_duration(tuplet)
```

```
>>> show(tuplet)
```



Returns none.

### 9.1.10 `labeltools.label_leaves_in_expr_with_leaf_durations`

`labeltools.label_leaves_in_expr_with_leaf_durations` (*expr*,  
*label\_durations=True*, *label\_written\_durations=True*,  
*markup\_direction=Down*)

Label leaves in expression with leaf durations.

**Example 1.** Label leaves with written durations:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> staff = stafftools.RhythmicStaff([tuplet])
>>> staff.override.text_script.staff_padding = 2.5
>>> staff.override.time_signature.stencil = False
>>> labeltools.label_leaves_in_expr_with_leaf_durations(
...     tuplet,
...     label_durations=False,
...     label_written_durations=True)
```

```
>>> show(staff)
```



**Example 2.** Label leaves with actual durations:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> staff = stafftools.RhythmicStaff([tuplet])
>>> staff.override.text_script.staff_padding = 2.5
>>> staff.override.time_signature.stencil = False
>>> labeltools.label_leaves_in_expr_with_leaf_durations(
...     tuplet,
...     label_durations=True,
...     label_written_durations=False)
```

```
>>> show(staff)
```

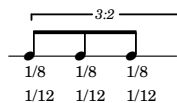


**Example 3.** Label leaves in tuplet with both written and actual durations:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> staff = stafftools.RhythmicStaff([tuplet])
>>> staff.override.text_script.staff_padding = 2.5
>>> staff.override.time_signature.stencil = False
>>> labeltools.label_leaves_in_expr_with_leaf_durations(
...     tuplet,
```

```
... label_durations=True,
... label_written_durations=True)
```

```
>>> show(staff)
```



Returns none.

### 9.1.11 `labeltools.label_leaves_in_expr_with_leaf_indices`

`labeltools.label_leaves_in_expr_with_leaf_indices` (*expr*,  
*markup\_direction=Down*)

Label leaves in *expr* with leaf indices:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_leaf_indices(staff)
>>> f(staff)
\new Staff {
  c'8 _ \markup { \small 0 }
  d'8 _ \markup { \small 1 }
  e'8 _ \markup { \small 2 }
  f'8 _ \markup { \small 3 }
}
```

```
>>> show(staff)
```



Returns none.

### 9.1.12 `labeltools.label_leaves_in_expr_with_leaf_numbers`

`labeltools.label_leaves_in_expr_with_leaf_numbers` (*expr*,  
*markup\_direction=Down*)

Label leaves in *expr* with leaf numbers:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_leaf_numbers(staff)
>>> f(staff)
\new Staff {
  c'8 _ \markup { \small 1 }
  d'8 _ \markup { \small 2 }
  e'8 _ \markup { \small 3 }
  f'8 _ \markup { \small 4 }
}
```

```
>>> show(staff)
```



Number leaves starting from 1.

Returns none.

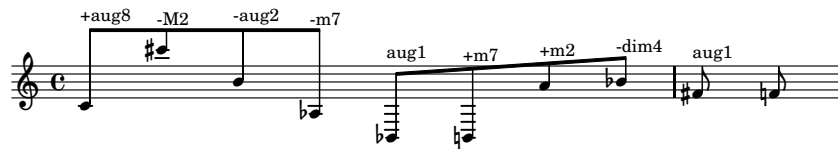
### 9.1.13 `labeltools.label_leaves_in_expr_with_named_interval_classes`

`labeltools.label_leaves_in_expr_with_named_interval_classes` (*expr*,  
*markup\_direction=Up*)

Label leaves in *expr* with named interval classes:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_named_interval_classes(staff)
```

```
>>> show(staff)
```



Returns none.

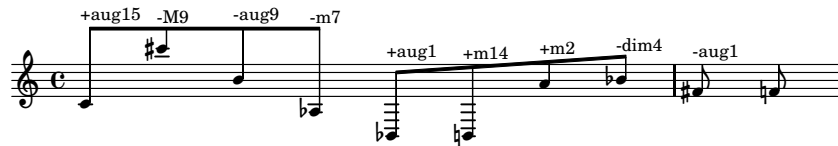
### 9.1.14 `labeltools.label_leaves_in_expr_with_named_intervals`

`labeltools.label_leaves_in_expr_with_named_intervals` (*expr*,  
*markup\_direction=Up*)

Label leaves in *expr* with named intervals:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_named_intervals(staff)
```

```
>>> show(staff)
```



Returns none.

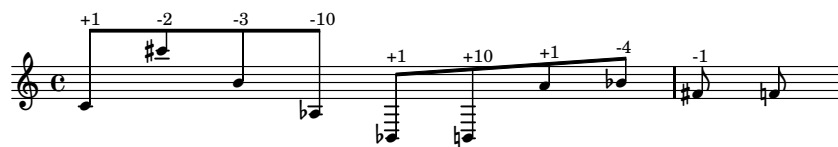
### 9.1.15 `labeltools.label_leaves_in_expr_with_numbered_interval_classes`

`labeltools.label_leaves_in_expr_with_numbered_interval_classes` (*expr*,  
*markup\_direction=Up*)

Label leaves in *expr* with numbered interval classes:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_numbered_interval_classes(staff)
```

```
>>> show(staff)
```



Returns none.

### 9.1.16 `labeltools.label_leaves_in_expr_with_numbered_intervals`

`labeltools.label_leaves_in_expr_with_numbered_intervals` (*expr*,  
*markup\_direction=Up*)

Label leaves in *expr* with numbered intervals:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_numbered_intervals(staff)
```

```
>>> show(staff)
```



Returns none.

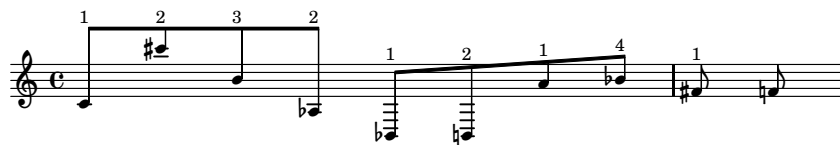
### 9.1.17 `labeltools.label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes`

`labeltools.label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes` (*expr*,  
*markup\_direction=Up*)

Label leaves in *expr* with numbered inversion-equivalent interval classes:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes(
...     staff)
```

```
>>> show(staff)
```



Returns none.

### 9.1.18 `labeltools.label_leaves_in_expr_with_pitch_class_numbers`

`labeltools.label_leaves_in_expr_with_pitch_class_numbers` (*expr*, *number=True*,  
*color=False*,  
*markup\_direction=Down*)

Label leaves in *expr* with pitch-class numbers:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_pitch_class_numbers(staff)
>>> print staff.lilypond_format
\new Staff {
  c'8 _ \markup { \small 0 }
  d'8 _ \markup { \small 2 }
  e'8 _ \markup { \small 4 }
  f'8 _ \markup { \small 5 }
}
```

```
>>> show(staff)
```



When `color=True` call `color_note_head_by_numbered_pitch_class_color_map()`:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_pitch_class_numbers(
...     staff, color=True, number=False)
>>> print staff.lilypond_format
\new Staff {
  \once \override NoteHead #'color = #(x11-color 'red)
  c'8
  \once \override NoteHead #'color = #(x11-color 'orange)
  d'8
  \once \override NoteHead #'color = #(x11-color 'ForestGreen)
  e'8
  \once \override NoteHead #'color = #(x11-color 'MediumOrchid)
  f'8
}
```

```
>>> show(staff)
```



You can set *number* and *color* at the same time.

Returns none.

### 9.1.19 labeltools.label\_leaves\_in\_expr\_with\_pitch\_numbers

`labeltools.label_leaves_in_expr_with_pitch_numbers` (*expr*,  
*markup\_direction=Down*)

Label leaves in *expr* with pitch numbers:

```
>>> staff = Staff(leaftools.make_leaves([None, 12, [13, 14, 15], None], [(1, 4)]))
>>> labeltools.label_leaves_in_expr_with_pitch_numbers(staff)
>>> f(staff)
\new Staff {
  r4
  c''4 _ \markup { \small 12 }
  <cs'' d'' ef''>4 _ \markup { \column { \small 15 \small 14 \small 13 } }
  r4
}
```

```
>>> show(staff)
```



Returns none.

### 9.1.20 labeltools.label\_leaves\_in\_expr\_with\_tuplet\_depth

`labeltools.label_leaves_in_expr_with_tuplet_depth` (*expr*,  
*markup\_direction=Down*)

Label leaves in *expr* with tuplet depth:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8")
>>> tuplettools.FixedDurationTuplet(Duration(2, 8), staff[-3:])
FixedDurationTuplet(1/4, [e'8, f'8, g'8])
>>> labeltools.label_leaves_in_expr_with_tuplet_depth(staff)
>>> f(staff)
\new Staff {
  c'8 _ \markup { \small 0 }
  d'8 _ \markup { \small 0 }
  \times 2/3 {
    e'8 _ \markup { \small 1 }
    f'8 _ \markup { \small 1 }
  }
```

```

    g'8 _ \markup { \small 1 }
  }
}

```

```
>>> show(staff)
```



Returns none.

### 9.1.21 `labeltools.label_leaves_in_expr_with_written_leaf_duration`

`labeltools.label_leaves_in_expr_with_written_leaf_duration` (*expr*,  
*markup\_direction=Down*)

Label leaves in *expr* with written leaf duration:

```

>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> labeltools.label_leaves_in_expr_with_written_leaf_durations(tuplet)
>>> f(tuplet)
\times 2/3 {
  c'8 _ \markup { \column { \small 1/8 \small 1/12 } }
  d'8 _ \markup { \column { \small 1/8 \small 1/12 } }
  e'8 _ \markup { \column { \small 1/8 \small 1/12 } }
}

```

```
>>> show(tuplet)
```



Returns none.

### 9.1.22 `labeltools.label_notes_in_expr_with_note_indices`

`labeltools.label_notes_in_expr_with_note_indices` (*expr*, *markup\_direction=Down*)

Label notes in *expr* with note indices:

```
>>> staff = Staff("c'8 d'8 r8 r8 g'8 a'8 r8 c''8")
```

```
>>> labeltools.label_notes_in_expr_with_note_indices(staff)
```

```
>>> show(staff)
```



Returns none.

### 9.1.23 `labeltools.label_tie_chains_in_expr_with_tie_chain_duration`

`labeltools.label_tie_chains_in_expr_with_tie_chain_duration` (*expr*,  
*markup\_direction=Down*)

Label tie chains in *expr* with tie chain durations:

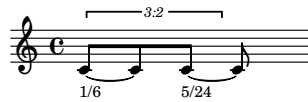
```

>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_tie_chains_in_expr_with_tie_chain_duration(staff)

```



```
>>> show(staff)
```



Returns none.

### 9.1.24 `labeltools.label_tie_chains_in_expr_with_tie_chain_durations`

`labeltools.label_tie_chains_in_expr_with_tie_chain_durations` (*expr*,  
*markup\_direction=Down*)  
 Label tie chains in *expr* with both written tie chain duration and prolated tie chain duration:

```
>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_tie_chains_in_expr_with_tie_chain_durations(staff)
```

```
>>> show(staff)
```



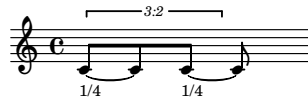
Returns none.

### 9.1.25 `labeltools.label_tie_chains_in_expr_with_written_tie_chain_duration`

`labeltools.label_tie_chains_in_expr_with_written_tie_chain_duration` (*expr*,  
*markup\_direction=Down*)  
 Label tie chains in *expr* with written tie chain duration:

```
>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_tie_chains_in_expr_with_written_tie_chain_duration(
...     staff)
```

```
>>> show(staff)
```



Returns none.

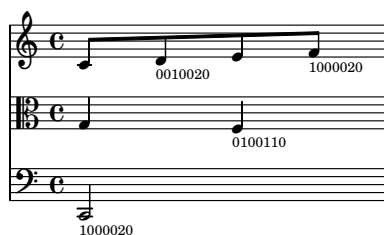
### 9.1.26 `labeltools.label_vertical_moments_in_expr_with_interval_class_vectors`

`labeltools.label_vertical_moments_in_expr_with_interval_class_vectors` (*expr*,  
*markup\_direction=Down*)  
 Label interval-class vector of every vertical moment in *expr*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_interval_class_vectors(score)
```

```
>>> show(score)
```



Returns none.

### 9.1.27 `labeltools.label_vertical_moments_in_expr_with_named_intervals`

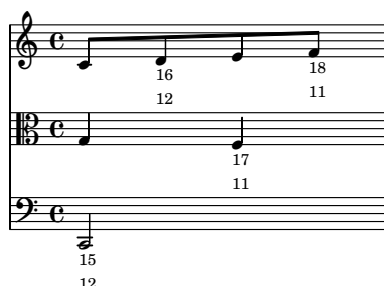
`labeltools.label_vertical_moments_in_expr_with_named_intervals` (*expr*,  
*markup\_direction=Down*)

Label named intervals of every vertical moment in *expr*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_named_intervals(score)
```

```
>>> show(score)
```



Returns none.

### 9.1.28 `labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes`

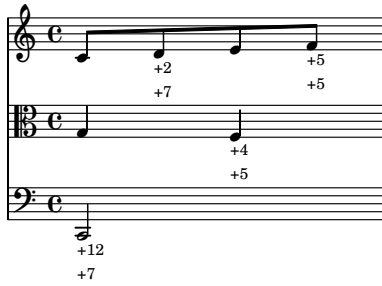
`labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes` (*expr*,  
*markup\_direction=Down*)

Label numbered interval-classes of every vertical moment in *expr*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes(
...     score)
```

```
>>> show(score)
```



Returns none.

### 9.1.29 `labeltools.label_vertical_moments_in_expr_with_numbered_intervals`

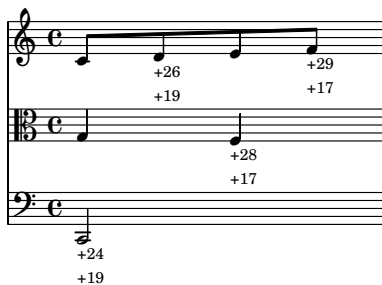
`labeltools.label_vertical_moments_in_expr_with_numbered_intervals` (*expr*,  
*markup\_direction=Down*)

Label numbered intervals of every vertical moment in *expr*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_numbered_intervals(
...     score)
```

```
>>> show(score)
```



Returns none.

### 9.1.30 `labeltools.label_vertical_moments_in_expr_with_numbered_pitch_classes`

`labeltools.label_vertical_moments_in_expr_with_numbered_pitch_classes` (*expr*,  
*markup\_direction=Down*)

Label pitch-classes of every vertical moment in *expr*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_numbered_pitch_classes(
...     score)
```

```
>>> show(score)
```



Returns none.

### 9.1.31 `labeltools.label_vertical_moments_in_expr_with_pitch_numbers`

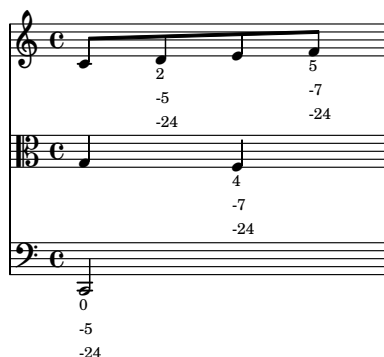
`labeltools.label_vertical_moments_in_expr_with_pitch_numbers` (*expr*,  
*markup\_direction=Down*)

Label pitch numbers of every vertical moment in *expr*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_pitch_numbers(score)
```

```
>>> show(score)
```



Returns none.

### 9.1.32 `labeltools.remove_markup_from_leaves_in_expr`

`labeltools.remove_markup_from_leaves_in_expr` (*expr*)

Remove markup from leaves in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_pitch_class_numbers(staff)
>>> f(staff)
\new Staff {
  c'8 _ \markup { \small 0 }
  d'8 _ \markup { \small 2 }
  e'8 _ \markup { \small 4 }
  f'8 _ \markup { \small 5 }
}
```

```
>>> show(staff)
```



```
>>> labeltools.remove_markup_from_leaves_in_expr(staff)
>>> f(staff)
\\new Staff {
  c'8
  d'8
  e'8
  f'8
}
```

```
>>> show(staff)
```



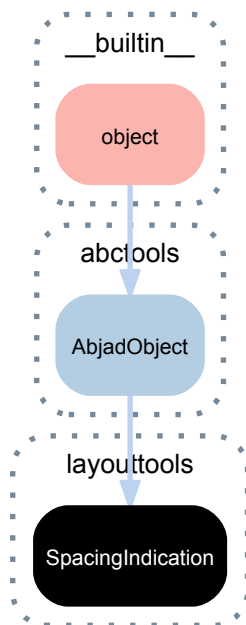
Returns none.



# LAYOUTTOOLS

## 10.1 Concrete classes

### 10.1.1 layouttools.SpacingIndication



**class** `layouttools.SpacingIndication(*args)`  
Spacing indication token.

`LilyPond` `Score.proportionalNotationDuration` will equal  
`proportional_notation_duration` when `tempo` equals `tempo_indication`.

Initialize from tempo mark and proportional notation duration:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 44)
>>> indication = layouttools.SpacingIndication(tempo, Duration(1, 68))
```

```
>>> indication
SpacingIndication(TempoMark(Duration(1, 8), 44), Duration(1, 68))
```

Initialize from constants:

```
>>> layouttools.SpacingIndication(((1, 8), 44), (1, 68))
SpacingIndication(TempoMark(Duration(1, 8), 44), Duration(1, 68))
```

Initialize from other spacing indication:

```
>>> layouttools.SpacingIndication(indication)
SpacingIndication(TempoMark(Duration(1, 8), 44), Duration(1, 68))
```

Spacing indications are immutable.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`SpacingIndication.normalized_spacing_duration`

Proportional notation duration at 60 MM.

`SpacingIndication.proportional_notation_duration`

LilyPond proportional notation duration context setting.

`SpacingIndication.tempo_indication`

Abjad tempo indication object.

## Special methods

`SpacingIndication.__eq__(expr)`

Spacing indications compare equal when normalized spacing durations compare equal.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

# 10.2 Functions

## 10.2.1 layouttools.make\_spacing\_vector

`layouttools.make_spacing_vector` (*basic\_distance*, *minimum\_distance*, *padding*, *stretchability*)

Make spacing vector:

```
>>> vector = layouttools.make_spacing_vector(0, 0, 12, 0)
```

Use to set paper block spacing attributes:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(staff)
>>> spacing_vector = layouttools.make_spacing_vector(0, 0, 12, 0)
>>> lilypond_file.paper_block.system_system_spacing = spacing_vector
```

Returns scheme vector.



### 10.2.2 layouttools.set\_line\_breaks\_by\_line\_duration

```
layouttools.set_line_breaks_by_line_duration(expr, line_duration,
                                             line_break_class=None,
                                             kind='prolated', adjust_eol=False,
                                             add_empty_bars=False)
```

Iterate *line\_break\_class* instances in *expr* and accumulate *kind* duration.

Add line break after every total less than or equal to *line\_duration*.

Set *line\_break\_class* to measure when *line\_break\_class* is none.

### 10.2.3 layouttools.set\_line\_breaks\_cyclically\_by\_line\_duration\_ge

```
layouttools.set_line_breaks_cyclically_by_line_duration_ge(expr,
                                                           line_duration,
                                                           line_break_class=None,
                                                           adjust_eol=False,
                                                           add_empty_bars=False)
```

Iterate *line\_break\_class* instances in *expr* and accumulate prolated duration.

Add line break after every total less than or equal to *line\_duration*:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 8), "c'8 d'8"))
>>> staff.append(Measure((2, 8), "e'8 f'8"))
>>> staff.append(Measure((2, 8), "g'8 a'8"))
>>> staff.append(Measure((2, 8), "b'8 c'8"))
>>> show(staff)
```



```
>>> layouttools.set_line_breaks_cyclically_by_line_duration_ge(
...     staff,
...     Duration(4, 8),
...     )
>>> show(staff)
```



```
>>> f(staff)
\new Staff {
  {
    \time 2/8
    c'8
    d'8
  }
  {
    e'8
    f'8
    \break
  }
  {
    g'8
    a'8
  }
  {
    b'8
    c'8
    \break
  }
}
```

When `line_break_class=None` set `line_break_class` to measure.

Set `adjust_eol` to `True` to include a magic Scheme incantation to move end-of-line LilyPond TimeSignature and BarLine grobs to the right.

### 10.2.4 layouttools.set\_line\_breaks\_cyclically\_by\_line\_duration\_in\_seconds\_ge

```
layouttools.set_line_breaks_cyclically_by_line_duration_in_seconds_ge (expr,
                                                                    line_duration,
                                                                    line_break_class=None,
                                                                    ad-
                                                                    just_eol=False,
                                                                    add_emptyBars=False)
```

Iterate `line_break_class` instances in `expr` and accumulate duration in seconds.

Add line break after every total less than or equal to `line_duration`:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 8), "c'8 d'8"))
>>> staff.append(Measure((2, 8), "e'8 f'8"))
>>> staff.append(Measure((2, 8), "g'8 a'8"))
>>> staff.append(Measure((2, 8), "b'8 c'8"))
>>> tempo_mark = contexttools.TempoMark(
...     Duration(1, 8), 44, target_context=Staff)
>>> tempo_mark = tempo_mark.attach(staff)
>>> show(staff)
```



```
>>> layouttools.set_line_breaks_cyclically_by_line_duration_in_seconds_ge(
...     staff, Duration(6))
>>> show(staff)
```



```
>>> f(staff)
\new Staff {
  \tempo 8=44
  {
    \time 2/8
    c'8
    d'8
  }
  {
    e'8
    f'8
    \break
  }
  {
    g'8
    a'8
  }
  {
    b'8
    c'8
  }
}
```

When `line_break_class=None` set `line_break_class` to measure.

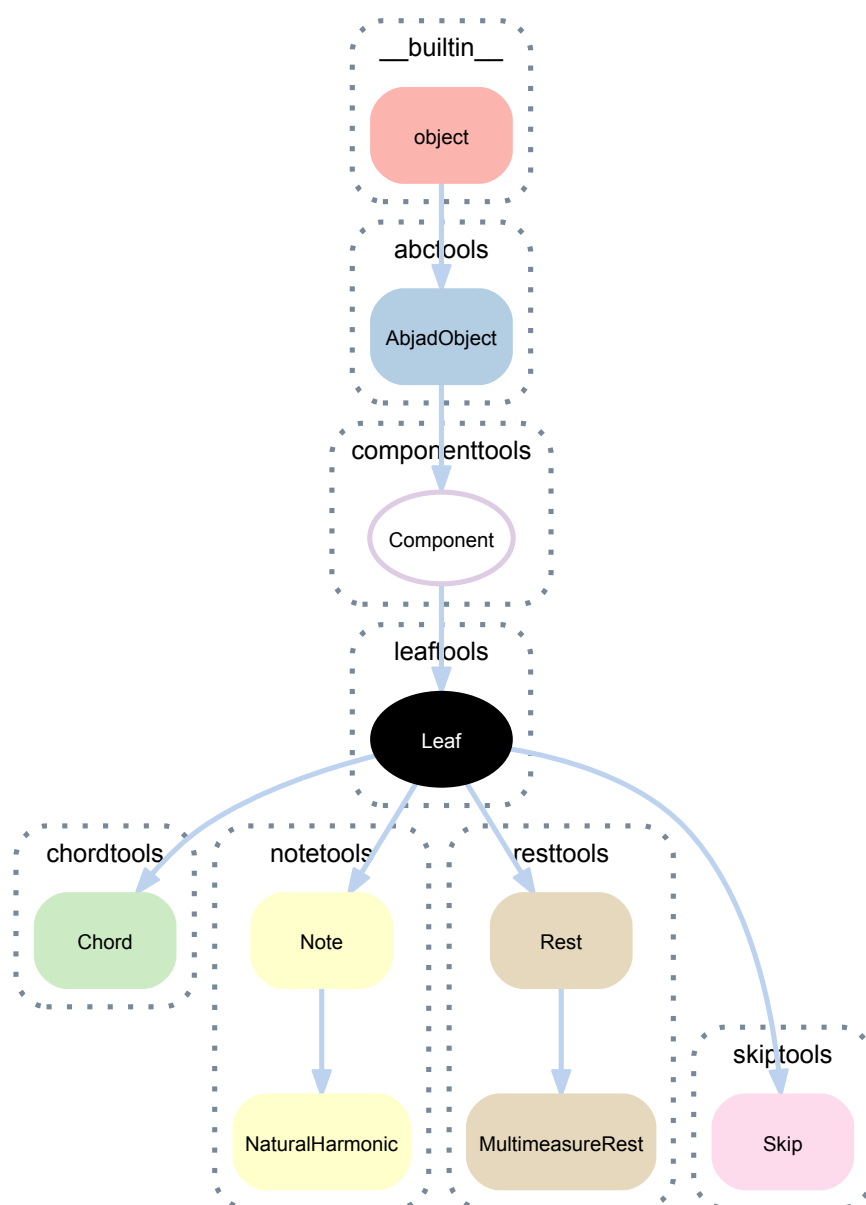
Set `adjust_eol = True` to include a magic Scheme incantation to move end-of-line LilyPond TimeSignature and BarLine grobs to the right.



# LEAFTOOLS

## 11.1 Abstract classes

### 11.1.1 leaftools.Leaf



**class** `leaftools.Leaf` (*written\_duration*, *lilypond\_duration\_multiplier=None*)  
Abstract base class for notes, rests, chords and skips.

### Bases

- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(Component).lilypond_format`  
Lilypond format of component.  
Returns string.

`(Component).override`  
LilyPond grob override component plug-in.  
Returns LilyPond grob override component plug-in.

`(Component).set`  
LilyPond context setting component plug-in.  
Returns LilyPond context setting component plug-in.

`(Component).storage_format`  
Storage format of component.  
Returns string.

### Read/write properties

`Leaf.lilypond_duration_multiplier`  
LilyPond duration multiplier.  
Set to positive multiplier or none.  
Returns positive multiplier or none.

`Leaf.written_duration`  
Written duration of leaf.  
Set to duration.  
Returns duration.

`Leaf.written_pitch_indication_is_at_sounding_pitch`  
Returns true when written pitch is at sounding pitch. Returns false when written pitch is transposed.

`Leaf.written_pitch_indication_is_nonsemantic`  
Returns true when pitch is nonsemantic. Returns false otherwise.  
Set to true when using leaves only graphically.  
Setting this value to true sets sounding pitch indicator to false.

## Methods

`(Component).select(sequential=False)`  
 Selects component.  
 Returns component selection when *sequential* is false.  
 Returns sequential selection when *sequential* is true.

## Special methods

`(Component).__copy__(*args)`  
 Copies component with marks but without children of component or spanners attached to component.  
 Returns new component.

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(Component).__mul__(n)`  
 Copies component *n* times and detaches spanners.  
 Returns list of new components.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`Leaf.__repr__()`  
 Interpreter representation of leaf.  
 Returns string.

`(Component).__rmul__(n)`  
 Copies component *n* times and detach spanners.  
 Returns list of new components.

`Leaf.__str__()`  
 String representation of leaf.  
 Returns string.

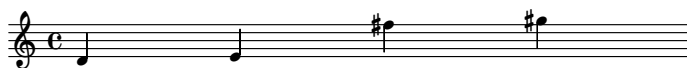
## 11.2 Functions

### 11.2.1 leaftools.make\_leaves

`leaftools.make_leaves(pitches, durations, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None, metrical_hierarchy=None)`  
 Make leaves.

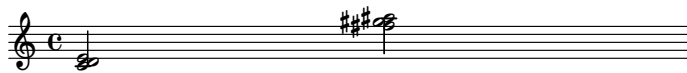
**Example 1.** Integer and string elements in *pitches* result in notes:

```
>>> pitches = [2, 4, 'F#5', 'G#5']
>>> duration = Duration(1, 4)
>>> leaves = leaftools.make_leaves(pitches, duration)
>>> staff = Staff(leaves)
>>> show(staff)
```



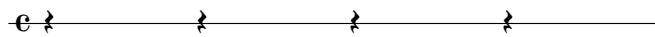
**Example 2.** Tuple elements in *pitches* result in chords:

```
>>> pitches = [(0, 2, 4), ('F#5', 'G#5', 'A#5')]
>>> duration = Duration(1, 2)
>>> leaves = leaftools.make_leaves(pitches, duration)
>>> staff = Staff(leaves)
>>> show(staff)
```



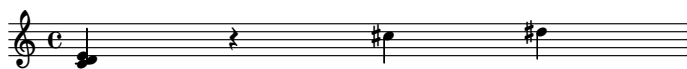
**Example 3.** None-valued elements in *pitches* result in rests:

```
>>> pitches = 4 * [None]
>>> durations = [Duration(1, 4)]
>>> leaves = leaftools.make_leaves(pitches, durations)
>>> staff = stafftools.RhythmicStaff(leaves)
>>> show(staff)
```



**Example 4.** You can mix and match values passed to *pitches*:

```
>>> pitches = [(0, 2, 4), None, 'C#5', 'D#5']
>>> durations = [Duration(1, 4)]
>>> leaves = leaftools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```



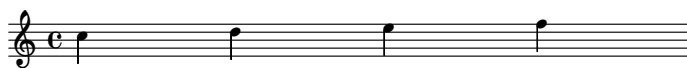
**Example 5.** Read *pitches* cyclically when the length of *pitches* is less than the length of *durations*:

```
>>> pitches = ['C5']
>>> durations = 2 * [Duration(3, 8), Duration(1, 8)]
>>> leaves = leaftools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```



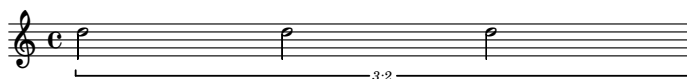
**Example 6.** Read *durations* cyclically when the length of *durations* is less than the length of *pitches*:

```
>>> pitches = "c' d' e' f'"
>>> durations = [Duration(1, 4)]
>>> leaves = leaftools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```



**Example 7.** Elements in *durations* with non-power-of-two denominators result in tuplet-nested leaves:

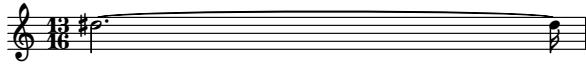
```
>>> pitches = ['D5']
>>> durations = [Duration(1, 3), Duration(1, 3), Duration(1, 3)]
>>> leaves = leaftools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```





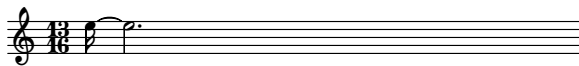
**Example 8.** Set *decrease\_durations\_monotonically* to true to return nonassignable durations tied from greatest to least:

```
>>> pitches = ['D#5']
>>> durations = [Duration(13, 16)]
>>> leaves = leaftools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((13, 16))
>>> time_signature = time_signature.attach(staff)
>>> show(staff)
```



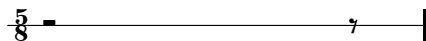
**Example 9.** Set *decrease\_durations\_monotonically* to false to return nonassignable durations tied from least to greatest:

```
>>> pitches = ['E5']
>>> durations = [Duration(13, 16)]
>>> leaves = leaftools.make_leaves(
...     pitches,
...     durations,
...     decrease_durations_monotonically=False,
... )
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((13, 16))
>>> time_signature = time_signature.attach(staff)
>>> show(staff)
```



**Example 10.** Set *tie\_rests* to true to return tied rests for nonassignable durations. Note that LilyPond does not engrave ties between rests:

```
>>> pitches = [None]
>>> durations = [Duration(5, 8)]
>>> leaves = leaftools.make_leaves(
...     pitches,
...     durations,
...     tie_rests=True,
... )
>>> staff = stafftools.RhythmicStaff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((5, 8))
>>> time_signature = time_signature.attach(staff)
>>> show(staff)
```



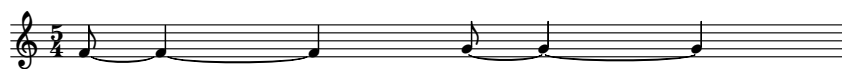
**Example 11.** Set *forbidden\_written\_duration* to avoid notes greater than or equal to a certain written duration:

```
>>> pitches = "f' g'"
>>> durations = [Duration(5, 8)]
>>> leaves = leaftools.make_leaves(
...     pitches,
...     durations,
...     forbidden_written_duration=Duration(1, 2),
... )
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))
>>> time_signature = time_signature.attach(staff)
>>> show(staff)
```



**Example 12.** You may set *forbidden\_written\_duration* and *decrease\_durations\_monotonically* together:

```
>>> pitches = "f' g'"
>>> durations = [Duration(5, 8)]
>>> leaves = leaftools.make_leaves(
...     pitches,
...     durations,
...     forbidden_written_duration=Duration(1, 2),
...     decrease_durations_monotonically=False,
... )
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))
>>> time_signature.attach(staff)
>>> show(staff)
```



Returns selection of unincorporated leaves.

### 11.2.2 leaftools.make\_leaves\_from\_talea

```
leaftools.make_leaves_from_talea(talea, talea_denominator, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None)
```

Make leaves from *talea*.

Interpret positive elements in *talea* as notes numerators.

Interpret negative elements in *talea* as rests numerators.

Set the pitch of all notes to middle C.

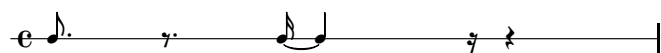
**Example 1.** Make leaves from talea:

```
>>> leaves = leaftools.make_leaves_from_talea([3, -3, 5, -5], 16)
>>> staff = stafftools.RhythmicStaff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((4, 4))(staff)
>>> show(staff)
```



**Example 2.** Increase durations monotonically:

```
>>> leaves = leaftools.make_leaves_from_talea(
...     [3, -3, 5, -5], 16,
...     decrease_durations_monotonically=False)
>>> staff = stafftools.RhythmicStaff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((4, 4))(staff)
>>> show(staff)
```



**Example 3.** Forbid written durations greater than or equal to a half note:

```
>>> leaves = leaftools.make_leaves_from_talea(
...     [3, -3, 5, -5], 16,
...     forbidden_written_duration=Duration(1, 4))
>>> staff = stafftools.RhythmicStaff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((4, 4))(staff)
>>> show(staff)
```



Returns list of leaves.

### 11.2.3 leaftools.make\_tied\_leaf

`leaftools.make_tied_leaf(kind, duration, decrease_durations_monotonically=True, forbidden_written_duration=None, pitches=None, tie_parts=True)`

Make tied *kind* with *duration*.

**Example 1.** Make note:

```
>>> leaves = leaftools.make_tied_leaf(
...     Note,
...     Duration(1, 2),
...     pitches='C#5',
... )
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((2, 4))(staff)
>>> show(staff)
```



**Example 2.** Make note and forbid half notes:

```
>>> leaves = leaftools.make_tied_leaf(
...     Note,
...     Duration(1, 2),
...     pitches='C#5',
...     forbidden_written_duration=Duration(1, 2),
... )
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((2, 4))(staff)
>>> show(staff)
```



**Example 3.** Make tied note with half notes forbidden and durations decreasing monotonically:

```
>>> leaves = leaftools.make_tied_leaf(
...     Note,
...     Duration(9, 8),
...     pitches='C#5',
...     forbidden_written_duration=Duration(1, 2),
...     decrease_durations_monotonically=True,
... )
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((9, 8))(staff)
>>> show(staff)
```



**Example 4.** Make tied note with half notes forbidden and durations increasing monotonically:

```
>>> leaves = leaftools.make_tied_leaf(
...     Note,
...     Duration(9, 8),
...     pitches='C#5',
...     forbidden_written_duration=Duration(1, 2),
...     decrease_durations_monotonically=False,
... )
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((9, 8))(staff)
>>> show(staff)
```



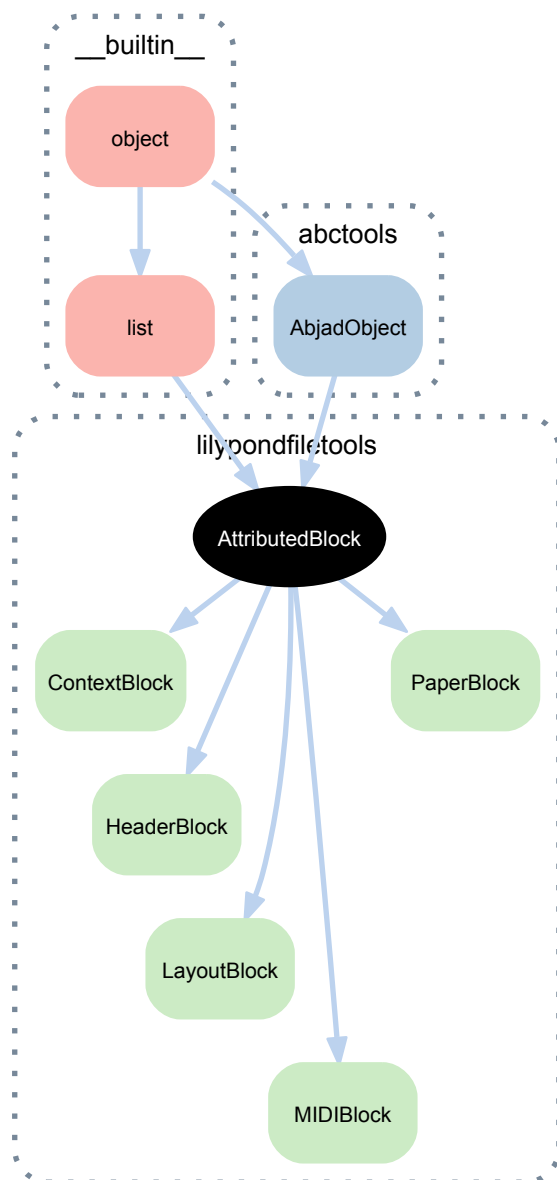
Returns selection of unincorporated leaves.



# LILYPONDFILETOOLS

## 12.1 Abstract classes

### 12.1.1 lilypondfiletools.AttributedBlock



`class lilypondfiletools.AttributedBlock`

Abjad model of LilyPond input file block with attributes.

## Bases

- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`AttributedBlock.lilypond_format`

## Read/write properties

`AttributedBlock.is_formatted_when_empty`

## Methods

- `(list).append()`  
L.append(object) – append object to end
- `(list).count(value)` → integer – return number of occurrences of value
- `(list).extend()`  
L.extend(iterable) – extend list by appending elements from the iterable
- `(list).index(value[, start[, stop]])` → integer – return first index of value.  
Raises `ValueError` if the value is not present.
- `(list).insert()`  
L.insert(index, object) – insert object before index
- `(list).pop([index])` → item – remove and return item at index (default last).  
Raises `IndexError` if list is empty or index is out of range.
- `(list).remove()`  
L.remove(value) – remove first occurrence of value. Raises `ValueError` if the value is not present.
- `(list).reverse()`  
L.reverse() – reverse *IN PLACE*
- `(list).sort()`  
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

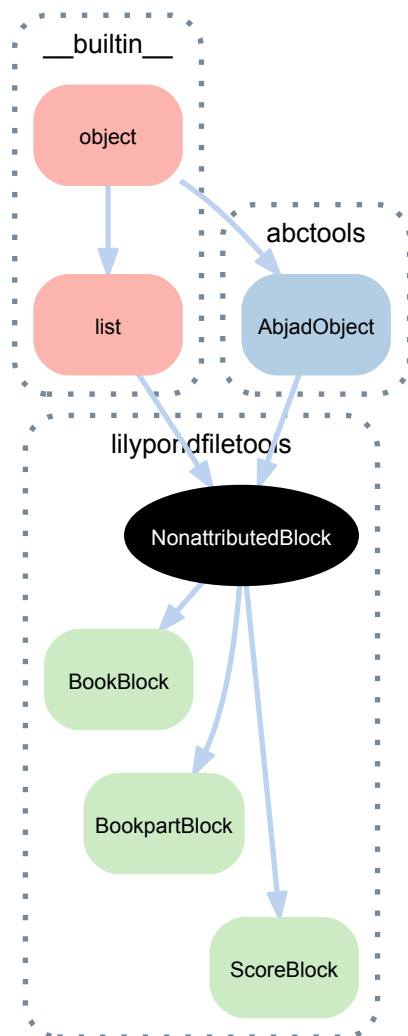
- `(list).__add__()`  
x.\_\_add\_\_(y) <==> x+y
- `(list).__contains__()`  
x.\_\_contains\_\_(y) <==> y in x
- `(list).__delitem__()`  
x.\_\_delitem\_\_(y) <==> del x[y]
- `(list).__delslice__()`  
x.\_\_delslice\_\_(i, j) <==> del x[i:j]  
Use of negative indices is not supported.

```

(list).__eq__()
    x.__eq__(y) <==> x==y
(list).__ge__()
    x.__ge__(y) <==> x>=y
(list).__getitem__()
    x.__getitem__(y) <==> x[y]
(list).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]
    Use of negative indices is not supported.
(list).__gt__()
    x.__gt__(y) <==> x>y
(list).__iadd__()
    x.__iadd__(y) <==> x+=y
(list).__imul__()
    x.__imul__(y) <==> x*=y
(list).__iter__() <==> iter(x)
(list).__le__()
    x.__le__(y) <==> x<=y
(list).__len__() <==> len(x)
(list).__lt__()
    x.__lt__(y) <==> x<y
(list).__mul__()
    x.__mul__(n) <==> x*n
(list).__ne__()
    x.__ne__(y) <==> x!=y
AttributedBlock.__repr__()
(list).__reversed__()
    L.__reversed__() – return a reverse iterator over the list
(list).__rmul__()
    x.__rmul__(n) <==> n*x
(list).__setitem__()
    x.__setitem__(i, y) <==> x[i]=y
(list).__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y
    Use of negative indices is not supported.

```

### 12.1.2 lilypondfiletools.NonattributedBlock



**class** `lilypondfiletools.NonattributedBlock`  
 Abjad model of LilyPond input file block with no attributes.

#### Bases

- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`NonattributedBlock.lilypond_format`

#### Read/write properties

`NonattributedBlock.is_formatted_when_empty`



## Methods

(list). **append**()  
L.append(object) – append object to end

(list). **count**(value) → integer – return number of occurrences of value

(list). **extend**()  
L.extend(iterable) – extend list by appending elements from the iterable

(list). **index**(value[, start[, stop]]) → integer – return first index of value.  
Raises ValueError if the value is not present.

(list). **insert**()  
L.insert(index, object) – insert object before index

(list). **pop**([index]) → item – remove and return item at index (default last).  
Raises IndexError if list is empty or index is out of range.

(list). **remove**()  
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

(list). **reverse**()  
L.reverse() – reverse *IN PLACE*

(list). **sort**()  
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

(list). **\_\_add\_\_**()  
x.\_\_add\_\_(y) <==> x+y

(list). **\_\_contains\_\_**()  
x.\_\_contains\_\_(y) <==> y in x

(list). **\_\_delitem\_\_**()  
x.\_\_delitem\_\_(y) <==> del x[y]

(list). **\_\_delslice\_\_**()  
x.\_\_delslice\_\_(i, j) <==> del x[i:j]  
Use of negative indices is not supported.

(list). **\_\_eq\_\_**()  
x.\_\_eq\_\_(y) <==> x==y

(list). **\_\_ge\_\_**()  
x.\_\_ge\_\_(y) <==> x>=y

(list). **\_\_getitem\_\_**()  
x.\_\_getitem\_\_(y) <==> x[y]

(list). **\_\_getslice\_\_**()  
x.\_\_getslice\_\_(i, j) <==> x[i:j]  
Use of negative indices is not supported.

(list). **\_\_gt\_\_**()  
x.\_\_gt\_\_(y) <==> x>y

(list). **\_\_iadd\_\_**()  
x.\_\_iadd\_\_(y) <==> x+=y

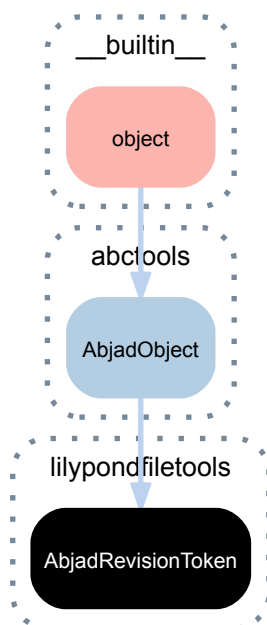
(list). **\_\_imul\_\_**()  
x.\_\_imul\_\_(y) <==> x\*=y

(list). **\_\_iter\_\_**() <==> iter(x)

```
(list).__le__()
    x.__le__(y) <==> x<=y
(list).__len__() <==> len(x)
(list).__lt__()
    x.__lt__(y) <==> x<y
(list).__mul__()
    x.__mul__(n) <==> x*n
(list).__ne__()
    x.__ne__(y) <==> x!=y
NonattributedBlock.__repr__()
(list).__reversed__()
    L.__reversed__() – return a reverse iterator over the list
(list).__rmul__()
    x.__rmul__(n) <==> n*x
(list).__setitem__()
    x.__setitem__(i, y) <==> x[i]=y
(list).__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y
    Use of negative indices is not supported.
```

## 12.2 Concrete classes

### 12.2.1 lilypondfiletools.AbjadRevisionToken



**class** lilypondfiletools.**AbjadRevisionToken**  
Abjad version token:

```
>>> lilypondfiletools.AbjadRevisionToken()
AbjadRevisionToken(Abjad revision ...)
```

Returns Abjad version token.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`AbjadRevisionToken.lilypond_format`  
Format contribution of Abjad version token:

```
>>> lilypondfiletools.AbjadRevisionToken().lilypond_format
'Abjad revision ...'
```

Returns string.

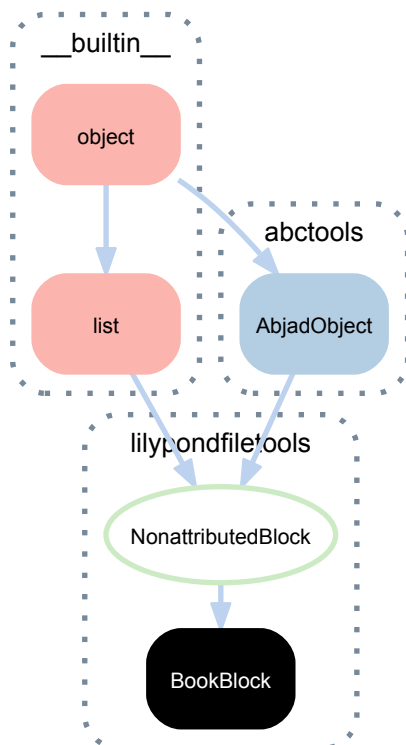
## Special methods

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`AbjadRevisionToken.__repr__()`

## 12.2.2 lilypondfiletools.BookBlock



**class** `lilypondfiletools.BookBlock`  
Abjad model of LilyPond input file book block:

```
>>> book_block = lilypondfiletools.BookBlock()
```

```
>>> book_block
BookBlock()
```

Returns book block.

## Bases

- `lilypondfiletools.NonattributedBlock`
- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(NonattributedBlock).lilypond_format`

## Read/write properties

`(NonattributedBlock).is_formatted_when_empty`

## Methods

`(list).append()`  
L.append(object) – append object to end

`(list).count(value)` → integer – return number of occurrences of value

`(list).extend()`  
L.extend(iterable) – extend list by appending elements from the iterable

`(list).index(value[, start[, stop]])` → integer – return first index of value.  
Raises `ValueError` if the value is not present.

`(list).insert()`  
L.insert(index, object) – insert object before index

`(list).pop([index])` → item – remove and return item at index (default last).  
Raises `IndexError` if list is empty or index is out of range.

`(list).remove()`  
L.remove(value) – remove first occurrence of value. Raises `ValueError` if the value is not present.

`(list).reverse()`  
L.reverse() – reverse *IN PLACE*

`(list).sort()`  
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

`(list).__add__()`  
x.\_\_add\_\_(y) <==> x+y

`(list).__contains__()`  
x.\_\_contains\_\_(y) <==> y in x

```

(list).__delitem__()
    x.__delitem__(y) <==> del x[y]

(list).__delslice__()
    x.__delslice__(i, j) <==> del x[i:j]

    Use of negative indices is not supported.

(list).__eq__()
    x.__eq__(y) <==> x==y

(list).__ge__()
    x.__ge__(y) <==> x>=y

(list).__getitem__()
    x.__getitem__(y) <==> x[y]

(list).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(list).__gt__()
    x.__gt__(y) <==> x>y

(list).__iadd__()
    x.__iadd__(y) <==> x+=y

(list).__imul__()
    x.__imul__(y) <==> x*=y

(list).__iter__() <==> iter(x)

(list).__le__()
    x.__le__(y) <==> x<=y

(list).__len__() <==> len(x)

(list).__lt__()
    x.__lt__(y) <==> x<y

(list).__mul__()
    x.__mul__(n) <==> x*n

(list).__ne__()
    x.__ne__(y) <==> x!=y

(NonattributedBlock).__repr__()

(list).__reversed__()
    L.__reversed__() – return a reverse iterator over the list

(list).__rmul__()
    x.__rmul__(n) <==> n*x

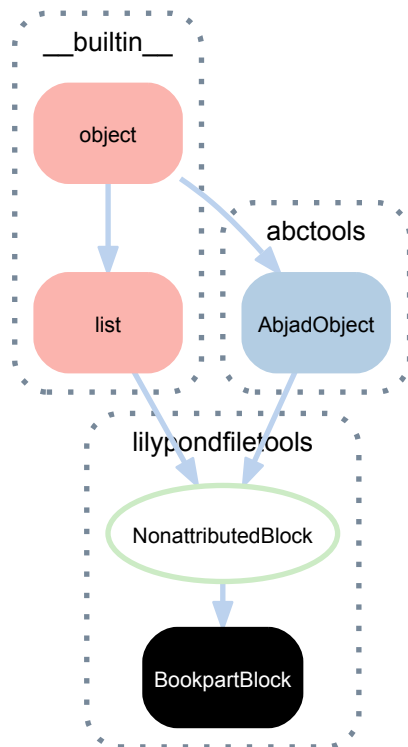
(list).__setitem__()
    x.__setitem__(i, y) <==> x[i]=y

(list).__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y

    Use of negative indices is not supported.

```

### 12.2.3 lilypondfiletools.BookpartBlock



**class** lilypondfiletools.**BookpartBlock**  
 Abjad model of LilyPond input file bookpart block:

```
>>> bookpart_block = lilypondfiletools.BookpartBlock()
```

```
>>> bookpart_block
BookpartBlock()
```

Returns bookpart block.

#### Bases

- lilypondfiletools.NonattributedBlock
- \_\_builtin\_\_.list
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

(NonattributedBlock).**lilypond\_format**

#### Read/write properties

(NonattributedBlock).**is\_formatted\_when\_empty**

#### Methods

(list).**append()**  
 L.append(object) – append object to end

(list).**count**(value) → integer – return number of occurrences of value

(list).**extend**()  
L.extend(iterable) – extend list by appending elements from the iterable

(list).**index**(value[, start[, stop]]) → integer – return first index of value.  
Raises ValueError if the value is not present.

(list).**insert**()  
L.insert(index, object) – insert object before index

(list).**pop**([index]) → item – remove and return item at index (default last).  
Raises IndexError if list is empty or index is out of range.

(list).**remove**()  
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

(list).**reverse**()  
L.reverse() – reverse *IN PLACE*

(list).**sort**()  
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

(list).**\_\_add\_\_**()  
x.\_\_add\_\_(y) <==> x+y

(list).**\_\_contains\_\_**()  
x.\_\_contains\_\_(y) <==> y in x

(list).**\_\_delitem\_\_**()  
x.\_\_delitem\_\_(y) <==> del x[y]

(list).**\_\_delslice\_\_**()  
x.\_\_delslice\_\_(i, j) <==> del x[i:j]  
  
Use of negative indices is not supported.

(list).**\_\_eq\_\_**()  
x.\_\_eq\_\_(y) <==> x==y

(list).**\_\_ge\_\_**()  
x.\_\_ge\_\_(y) <==> x>=y

(list).**\_\_getitem\_\_**()  
x.\_\_getitem\_\_(y) <==> x[y]

(list).**\_\_getslice\_\_**()  
x.\_\_getslice\_\_(i, j) <==> x[i:j]  
  
Use of negative indices is not supported.

(list).**\_\_gt\_\_**()  
x.\_\_gt\_\_(y) <==> x>y

(list).**\_\_iadd\_\_**()  
x.\_\_iadd\_\_(y) <==> x+=y

(list).**\_\_imul\_\_**()  
x.\_\_imul\_\_(y) <==> x\*=y

(list).**\_\_iter\_\_**() <==> iter(x)

(list).**\_\_le\_\_**()  
x.\_\_le\_\_(y) <==> x<=y

(list).**\_\_len\_\_**() <==> len(x)

```
(list).__lt__()
    x.__lt__(y) <==> x<y

(list).__mul__()
    x.__mul__(n) <==> x*n

(list).__ne__()
    x.__ne__(y) <==> x!=y

(NonattributedBlock).__repr__()

(list).__reversed__()
    L.__reversed__() – return a reverse iterator over the list

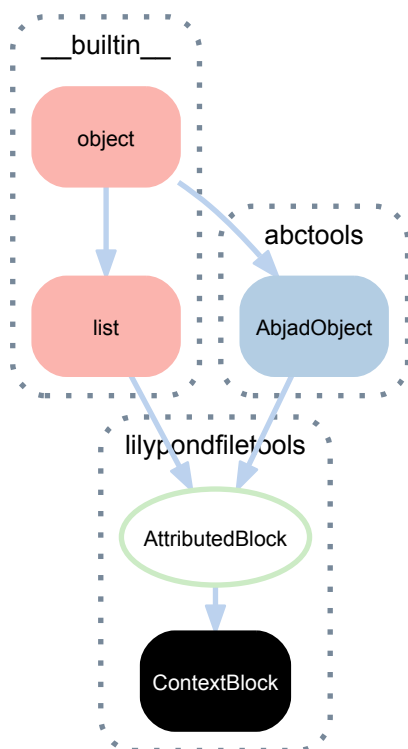
(list).__rmul__()
    x.__rmul__(n) <==> n*x

(list).__setitem__()
    x.__setitem__(i, y) <==> x[i]=y

(list).__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y

    Use of negative indices is not supported.
```

## 12.2.4 lilypondfiletools.ContextBlock



**class** `lilypondfiletools.ContextBlock` (*context\_name=None*)  
 Abjad model of LilyPond input file context block:

```
>>> context_block = lilypondfiletools.ContextBlock()
```

```
>>> context_block
ContextBlock()
```

```
>>> context_block.context_name = 'Score'
>>> context_block.override.bar_number.transparent = True
>>> scheme = schemetools.Scheme('end-of-line-invisible')
>>> context_block.override.time_signature.break_visibility = scheme
```



```
>>> context_block.set.proportionalNotationDuration = \
...     schemetools.SchemeMoment((1, 45))
```

Returns context block.

## Bases

- `lilypondfiletools.AttributedBlock`
- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ContextBlock.accepts`

`ContextBlock.engraver_consists`

`ContextBlock.engraver_removals`

`(AttributedBlock).lilypond_format`

`ContextBlock.override`  
Reference to LilyPond grob override component plug-in.

`ContextBlock.set`  
Reference LilyPond context setting component plug-in.

## Read/write properties

`ContextBlock.alias`  
Read / write alias.

`ContextBlock.context_name`  
Read / write context name.

`(AttributedBlock).is_formatted_when_empty`

`ContextBlock.name`  
Read / write name.

`ContextBlock.type`  
Read / write type.

## Methods

`(list).append()`  
L.append(object) – append object to end

`(list).count(value) → integer` – return number of occurrences of value

`(list).extend()`  
L.extend(iterable) – extend list by appending elements from the iterable

`(list).index(value[, start[, stop]]) → integer` – return first index of value.  
Raises `ValueError` if the value is not present.

`(list).insert()`  
L.insert(index, object) – insert object before index

(list). **pop** ([*index*]) → item – remove and return item at index (default last).  
 Raises IndexError if list is empty or index is out of range.

(list). **remove** ()  
 L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

(list). **reverse** ()  
 L.reverse() – reverse *IN PLACE*

(list). **sort** ()  
 L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

(list). **\_\_add\_\_** ()  
 x.\_\_add\_\_(y) <==> x+y

(list). **\_\_contains\_\_** ()  
 x.\_\_contains\_\_(y) <==> y in x

(list). **\_\_delitem\_\_** ()  
 x.\_\_delitem\_\_(y) <==> del x[y]

(list). **\_\_delslice\_\_** ()  
 x.\_\_delslice\_\_(i, j) <==> del x[i:j]  
 Use of negative indices is not supported.

(list). **\_\_eq\_\_** ()  
 x.\_\_eq\_\_(y) <==> x==y

(list). **\_\_ge\_\_** ()  
 x.\_\_ge\_\_(y) <==> x>=y

(list). **\_\_getitem\_\_** ()  
 x.\_\_getitem\_\_(y) <==> x[y]

(list). **\_\_getslice\_\_** ()  
 x.\_\_getslice\_\_(i, j) <==> x[i:j]  
 Use of negative indices is not supported.

(list). **\_\_gt\_\_** ()  
 x.\_\_gt\_\_(y) <==> x>y

(list). **\_\_iadd\_\_** ()  
 x.\_\_iadd\_\_(y) <==> x+=y

(list). **\_\_imul\_\_** ()  
 x.\_\_imul\_\_(y) <==> x\*=y

(list). **\_\_iter\_\_** () <==> iter(x)

(list). **\_\_le\_\_** ()  
 x.\_\_le\_\_(y) <==> x<=y

(list). **\_\_len\_\_** () <==> len(x)

(list). **\_\_lt\_\_** ()  
 x.\_\_lt\_\_(y) <==> x<y

(list). **\_\_mul\_\_** ()  
 x.\_\_mul\_\_(n) <==> x\*n

(list). **\_\_ne\_\_** ()  
 x.\_\_ne\_\_(y) <==> x!=y

(AttributedBlock). **\_\_repr\_\_** ()

```
(list).__reversed__()
    L.__reversed__() – return a reverse iterator over the list

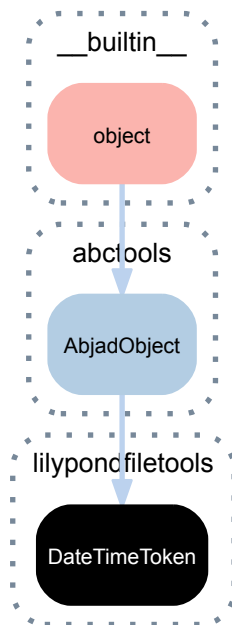
(list).__rmul__()
    x.__rmul__(n) <==> n*x

(list).__setitem__()
    x.__setitem__(i, y) <==> x[i]=y

(list).__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y

    Use of negative indices is not supported.
```

### 12.2.5 lilypondfiletools.DateTimeToken



**class** lilypondfiletools.DateTimeToken  
Date time token:

```
>>> lilypondfiletools.DateTimeToken()
DateTimeToken(...)
```

Returns date / time token.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`DateTimeToken.lilypond_format`  
Format contribution of date time token:

```
>>> lilypondfiletools.DateTimeToken().lilypond_format
'...'
```

Returns string.

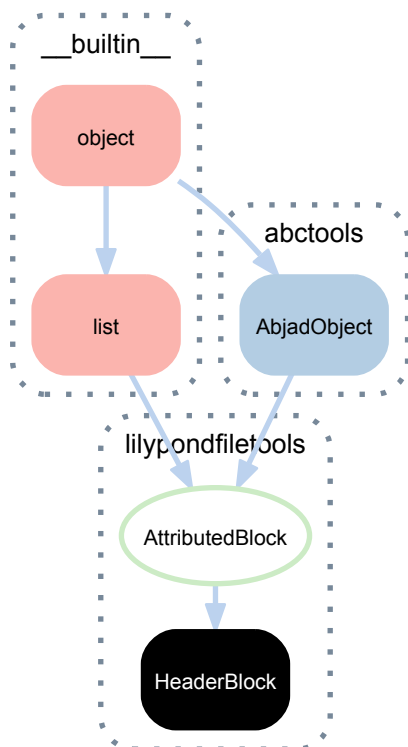
## Special methods

(AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

DateTimeToken.**\_\_repr\_\_**()

### 12.2.6 lilypondfiletools.HeaderBlock



**class** lilypondfiletools.**HeaderBlock**  
 Abjad model of LilyPond input file header block:

```
>>> header_block = lilypondfiletools.HeaderBlock()
```

```
>>> header_block
HeaderBlock()
```

```
>>> header_block.composer = markuptools.Markup('Josquin')
>>> header_block.title = markuptools.Markup('Missa sexti tonus')
```

Returns header block.

## Bases

- lilypondfiletools.AttributedBlock
- \_\_builtin\_\_.list
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

`(AttributedBlock).lilypond_format`

## Read/write properties

`(AttributedBlock).is_formatted_when_empty`

## Methods

`(list).append()`  
 L.append(object) – append object to end

`(list).count(value)` → integer – return number of occurrences of value

`(list).extend()`  
 L.extend(iterable) – extend list by appending elements from the iterable

`(list).index(value[, start[, stop]])` → integer – return first index of value.  
 Raises `ValueError` if the value is not present.

`(list).insert()`  
 L.insert(index, object) – insert object before index

`(list).pop([index])` → item – remove and return item at index (default last).  
 Raises `IndexError` if list is empty or index is out of range.

`(list).remove()`  
 L.remove(value) – remove first occurrence of value. Raises `ValueError` if the value is not present.

`(list).reverse()`  
 L.reverse() – reverse *IN PLACE*

`(list).sort()`  
 L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

`(list).__add__()`  
 x.\_\_add\_\_(y) <==> x+y

`(list).__contains__()`  
 x.\_\_contains\_\_(y) <==> y in x

`(list).__delitem__()`  
 x.\_\_delitem\_\_(y) <==> del x[y]

`(list).__delslice__()`  
 x.\_\_delslice\_\_(i, j) <==> del x[i:j]  
 Use of negative indices is not supported.

`(list).__eq__()`  
 x.\_\_eq\_\_(y) <==> x==y

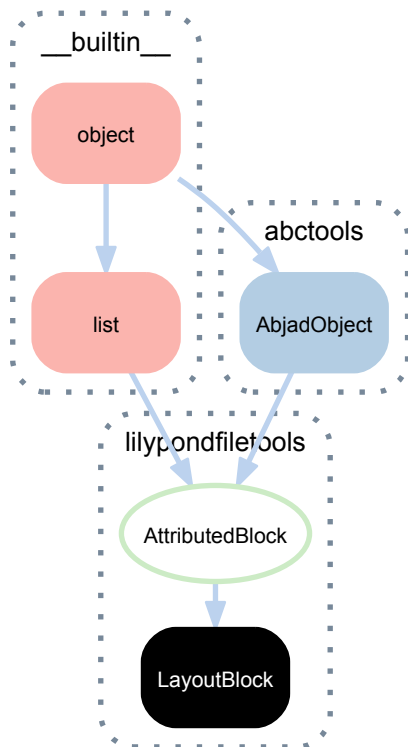
`(list).__ge__()`  
 x.\_\_ge\_\_(y) <==> x>=y

`(list).__getitem__()`  
 x.\_\_getitem\_\_(y) <==> x[y]

`(list).__getslice__()`  
 x.\_\_getslice\_\_(i, j) <==> x[i:j]  
 Use of negative indices is not supported.

```
(list) .__gt__()
    x.__gt__(y) <==> x>y
(list) .__iadd__()
    x.__iadd__(y) <==> x+=y
(list) .__imul__()
    x.__imul__(y) <==> x*=y
(list) .__iter__() <==> iter(x)
(list) .__le__()
    x.__le__(y) <==> x<=y
(list) .__len__() <==> len(x)
(list) .__lt__()
    x.__lt__(y) <==> x<y
(list) .__mul__()
    x.__mul__(n) <==> x*n
(list) .__ne__()
    x.__ne__(y) <==> x!=y
(AttributedBlock) .__repr__()
(list) .__reversed__()
    L.__reversed__() – return a reverse iterator over the list
(list) .__rmul__()
    x.__rmul__(n) <==> n*x
(list) .__setitem__()
    x.__setitem__(i, y) <==> x[i]=y
(list) .__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y
    Use of negative indices is not supported.
```

### 12.2.7 lilypondfiletools.LayoutBlock



**class** `lilypondfiletools.LayoutBlock`  
 Abjad model of LilyPond input file layout block:

```
>>> layout_block = lilypondfiletools.LayoutBlock()
```

```
>>> layout_block
LayoutBlock()
```

```
>>> layout_block.indent = 0
>>> layout_block.ragged_right = True
```

Returns layout block.

#### Bases

- `lilypondfiletools.AttributedBlock`
- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`LayoutBlock.context_blocks`  
 List of context blocks:

```
>>> layout_block = lilypondfiletools.LayoutBlock()
```

```
>>> context_block = lilypondfiletools.ContextBlock('Score')
>>> context_block.override.bar_number.transparent = True
```

```
>>> scheme_expr = schemetools.Scheme('end-of-line-invisible')
>>> context_block.override.time_signature.break_visibility = \
...     scheme_expr
>>> layout_block.context_blocks.append(context_block)
```

Returns list.

`LayoutBlock.contexts`

DEPRECATED. USE `CONTEXT_BLOCKS` INSTEAD.

`(AttributedBlock).lilypond_format`

## Read/write properties

`(AttributedBlock).is_formatted_when_empty`

## Methods

`(list).append()`

`L.append(object)` – append object to end

`(list).count(value)` → integer – return number of occurrences of value

`(list).extend()`

`L.extend(iterable)` – extend list by appending elements from the iterable

`(list).index(value[, start[, stop]])` → integer – return first index of value.

Raises `ValueError` if the value is not present.

`(list).insert()`

`L.insert(index, object)` – insert object before index

`(list).pop([index])` → item – remove and return item at index (default last).

Raises `IndexError` if list is empty or index is out of range.

`(list).remove()`

`L.remove(value)` – remove first occurrence of value. Raises `ValueError` if the value is not present.

`(list).reverse()`

`L.reverse()` – reverse *IN PLACE*

`(list).sort()`

`L.sort(cmp=None, key=None, reverse=False)` – stable sort *IN PLACE*; `cmp(x, y) -> -1, 0, 1`

## Special methods

`(list).__add__()`

`x.__add__(y)` <==> `x+y`

`(list).__contains__()`

`x.__contains__(y)` <==> `y in x`

`(list).__delitem__()`

`x.__delitem__(y)` <==> `del x[y]`

`(list).__delslice__()`

`x.__delslice__(i, j)` <==> `del x[i:j]`

Use of negative indices is not supported.

`(list).__eq__()`

`x.__eq__(y)` <==> `x==y`

`(list).__ge__()`

`x.__ge__(y)` <==> `x>=y`



```

(list).__getitem__()
    x.__getitem__(y) <==> x[y]

(list).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(list).__gt__()
    x.__gt__(y) <==> x>y

(list).__iadd__()
    x.__iadd__(y) <==> x+=y

(list).__imul__()
    x.__imul__(y) <==> x*=y

(list).__iter__() <==> iter(x)

(list).__le__()
    x.__le__(y) <==> x<=y

(list).__len__() <==> len(x)

(list).__lt__()
    x.__lt__(y) <==> x<y

(list).__mul__()
    x.__mul__(n) <==> x*n

(list).__ne__()
    x.__ne__(y) <==> x!=y

(AttributedBlock).__repr__()

(list).__reversed__()
    L.__reversed__() – return a reverse iterator over the list

(list).__rmul__()
    x.__rmul__(n) <==> n*x

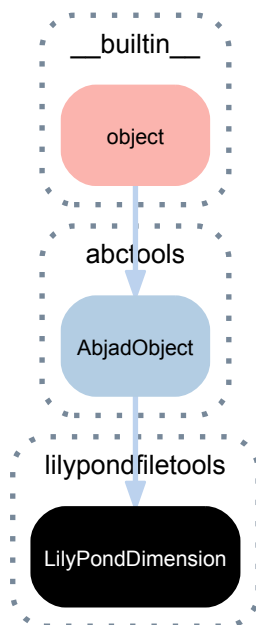
(list).__setitem__()
    x.__setitem__(i, y) <==> x[i]=y

(list).__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y

    Use of negative indices is not supported.

```

## 12.2.8 lilypondfiletools.LilyPondDimension



**class** `lilypondfiletools.LilyPondDimension` (*value*, *unit*)  
Abjad model of page dimensions in LilyPond:

```
>>> dimension = lilypondfiletools.LilyPondDimension(2, 'in')
```

Returns LilyPondDimension instance.

### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`LilyPondDimension.lilypond_format`

`LilyPondDimension.unit`

`LilyPondDimension.value`

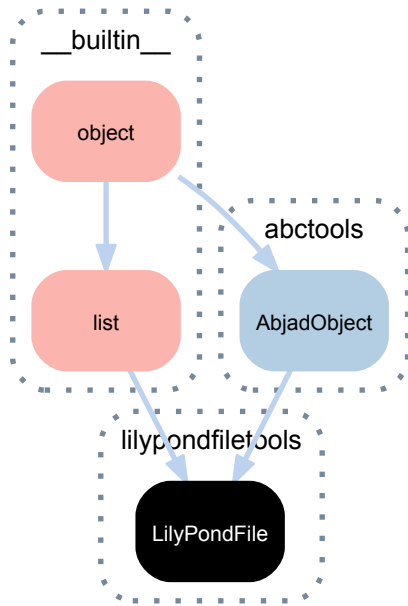
### Special methods

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

### 12.2.9 lilypondfiletools.LilyPondFile



**class** lilypondfiletools.**LilyPondFile**

Abjad model of LilyPond input file:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(staff)
>>> lilypond_file.file_initial_user_comments.append(
...     'File construct as an example.')
>>> lilypond_file.file_initial_user_comments.append(
...     'Parts shown here for positioning.')
>>> lilypond_file.file_initial_user_includes.append(
...     'external-settings-file-1.ly')
>>> lilypond_file.file_initial_user_includes.append(
...     'external-settings-file-2.ly')
>>> lilypond_file.default_paper_size = 'letter', 'portrait'
>>> lilypond_file.global_staff_size = 16
>>> lilypond_file.header_block.composer = \
...     markuptools.Markup('Josquin')
>>> lilypond_file.header_block.title = \
...     markuptools.Markup('Missa sexti tonus')
>>> lilypond_file.layout_block.indent = 0
>>> lilypond_file.layout_block.left_margin = 15
>>> lilypond_file.paper_block.oddFooterMarkup = \
...     markuptools.Markup('The odd-page footer')
>>> lilypond_file.paper_block.evenFooterMarkup = \
...     markuptools.Markup('The even-page footer')
```

#### Bases

- abctools.AbjadObject
- \_\_builtin\_\_.list
- \_\_builtin\_\_.object

#### Read-only properties

**LilyPondFile.lilypond\_format**

Format-time contribution of LilyPond file.

## Read/write properties

`LilyPondFile.default_paper_size`  
 LilyPond default paper size.

`LilyPondFile.file_initial_system_comments`  
 List of file-initial system comments.

`LilyPondFile.file_initial_system_includes`  
 List of file-initial system include commands.

`LilyPondFile.file_initial_user_comments`  
 List of file-initial user comments.

`LilyPondFile.file_initial_user_includes`  
 List of file-initial user include commands.

`LilyPondFile.global_staff_size`  
 LilyPond global staff size.

## Methods

`(list).append()`  
 L.append(object) – append object to end

`(list).count(value)` → integer – return number of occurrences of value

`(list).extend()`  
 L.extend(iterable) – extend list by appending elements from the iterable

`(list).index(value[, start[, stop]])` → integer – return first index of value.  
 Raises `ValueError` if the value is not present.

`(list).insert()`  
 L.insert(index, object) – insert object before index

`(list).pop([index])` → item – remove and return item at index (default last).  
 Raises `IndexError` if list is empty or index is out of range.

`(list).remove()`  
 L.remove(value) – remove first occurrence of value. Raises `ValueError` if the value is not present.

`(list).reverse()`  
 L.reverse() – reverse *IN PLACE*

`(list).sort()`  
 L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) → -1, 0, 1

## Special methods

`(list).__add__()`  
 x.\_\_add\_\_(y) <==> x+y

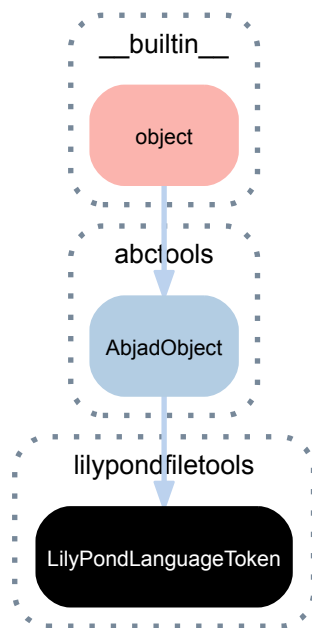
`(list).__contains__()`  
 x.\_\_contains\_\_(y) <==> y in x

`(list).__delitem__()`  
 x.\_\_delitem\_\_(y) <==> del x[y]

`(list).__delslice__()`  
 x.\_\_delslice\_\_(i, j) <==> del x[i:j]  
 Use of negative indices is not supported.



### 12.2.10 lilypondfiletools.LilyPondLanguageToken



**class** lilypondfiletools.LilyPondLanguageToken  
LilyPond language token:

```
>>> lilypondfiletools.LilyPondLanguageToken()
LilyPondLanguageToken('english')
```

Returns LilyPond language token.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`LilyPondLanguageToken.lilypond_format`  
Format contribution of LilyPond language token:

```
>>> lilypondfiletools.LilyPondLanguageToken().lilypond_format
'\\language "english"'
```

Returns string.

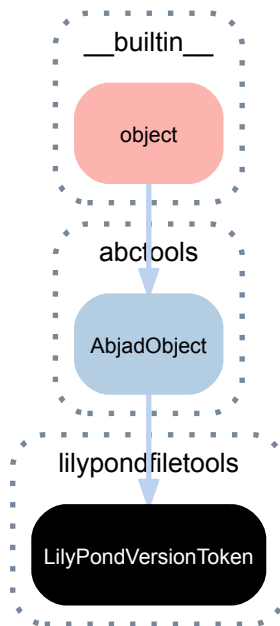
#### Special methods

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`LilyPondLanguageToken.__repr__()`

### 12.2.11 lilypondfiletools.LilyPondVersionToken



**class** lilypondfiletools.LilyPondVersionToken (*version=None*)  
 LilyPond version token:

```
>>> lilypondfiletools.LilyPondVersionToken()
LilyPondVersionToken(\version "...")
```

A specific version can also be specified:

Returns LilyPond version token.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`LilyPondVersionToken.lilypond_format`  
 Format contribution of LilyPond version token:

```
>>> lilypondfiletools.LilyPondVersionToken().lilypond_format
'\version "..."
```

Returns string.

`LilyPondVersionToken.version`

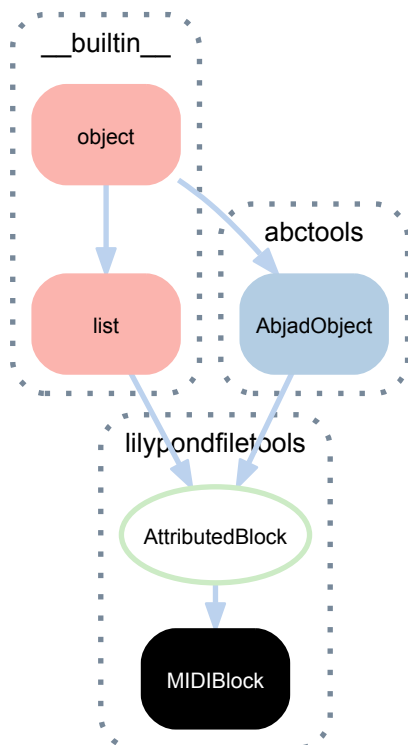
#### Special methods

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

```
LilyPondVersionToken.__repr__()
```

## 12.2.12 lilypondfiletools.MIDIBlock



**class lilypondfiletools.MIDIBlock**  
 Abjad model of LilyPond input file MIDI block:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> score = Score([staff])
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
```

```
>>> lilypond_file.score_block.append(lilypondfiletools.MIDIBlock())
```

```
>>> layout_block = lilypondfiletools.LayoutBlock()
>>> layout_block.is_formatted_when_empty = True
>>> lilypond_file.score_block.append(layout_block)
```

MIDI blocks are formatted even when they are empty.

The example here appends MIDI and layout blocks to a score block. Doing this allows LilyPond to create both MIDI and PDF output from a single input file.

Read the LilyPond docs on LilyPond file structure for the details as to why this is the case.

### Bases

- `lilypondfiletools.AttributedBlock`
- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`



## Read-only properties

`(AttributedBlock).lilypond_format`

## Read/write properties

`(AttributedBlock).is_formatted_when_empty`

## Methods

`(list).append()`  
 L.append(object) – append object to end

`(list).count(value)` → integer – return number of occurrences of value

`(list).extend()`  
 L.extend(iterable) – extend list by appending elements from the iterable

`(list).index(value[, start[, stop]])` → integer – return first index of value.  
 Raises `ValueError` if the value is not present.

`(list).insert()`  
 L.insert(index, object) – insert object before index

`(list).pop([index])` → item – remove and return item at index (default last).  
 Raises `IndexError` if list is empty or index is out of range.

`(list).remove()`  
 L.remove(value) – remove first occurrence of value. Raises `ValueError` if the value is not present.

`(list).reverse()`  
 L.reverse() – reverse *IN PLACE*

`(list).sort()`  
 L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

`(list).__add__()`  
 x.\_\_add\_\_(y) <==> x+y

`(list).__contains__()`  
 x.\_\_contains\_\_(y) <==> y in x

`(list).__delitem__()`  
 x.\_\_delitem\_\_(y) <==> del x[y]

`(list).__delslice__()`  
 x.\_\_delslice\_\_(i, j) <==> del x[i:j]  
 Use of negative indices is not supported.

`(list).__eq__()`  
 x.\_\_eq\_\_(y) <==> x==y

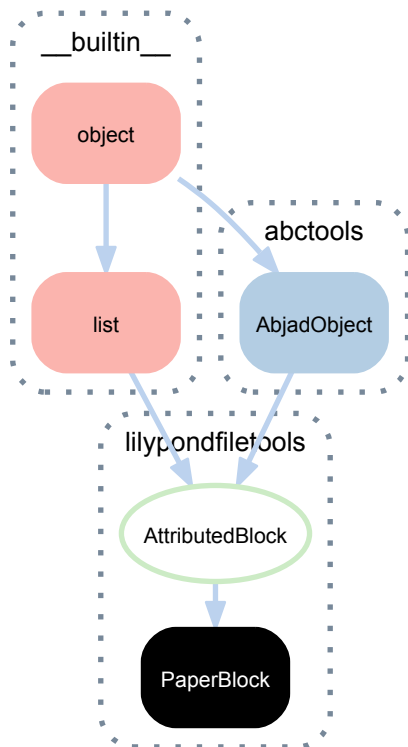
`(list).__ge__()`  
 x.\_\_ge\_\_(y) <==> x>=y

`(list).__getitem__()`  
 x.\_\_getitem\_\_(y) <==> x[y]

`(list).__getslice__()`  
 x.\_\_getslice\_\_(i, j) <==> x[i:j]  
 Use of negative indices is not supported.

```
(list) .__gt__()
    x.__gt__(y) <==> x>y
(list) .__iadd__()
    x.__iadd__(y) <==> x+=y
(list) .__imul__()
    x.__imul__(y) <==> x*=y
(list) .__iter__() <==> iter(x)
(list) .__le__()
    x.__le__(y) <==> x<=y
(list) .__len__() <==> len(x)
(list) .__lt__()
    x.__lt__(y) <==> x<y
(list) .__mul__()
    x.__mul__(n) <==> x*n
(list) .__ne__()
    x.__ne__(y) <==> x!=y
(AttributedBlock) .__repr__()
(list) .__reversed__()
    L.__reversed__() – return a reverse iterator over the list
(list) .__rmul__()
    x.__rmul__(n) <==> n*x
(list) .__setitem__()
    x.__setitem__(i, y) <==> x[i]=y
(list) .__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y
    Use of negative indices is not supported.
```

### 12.2.13 lilypondfiletools.PaperBlock



**class** `lilypondfiletools.PaperBlock`  
 Abjad model of LilyPond input file paper block:

```
>>> paper_block = lilypondfiletools.PaperBlock()
```

```
>>> paper_block
PaperBlock()
```

```
>>> paper_block.print_page_number = True
>>> paper_block.print_first_page_number = False
```

Returns paper block.

#### Bases

- `lilypondfiletools.AttributedBlock`
- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(AttributedBlock).lilypond_format`

#### Read/write properties

`(AttributedBlock).is_formatted_when_empty`

`PaperBlock.minimal_page_breaking`

## Methods

(list) **.append()**  
L.append(object) – append object to end

(list) **.count** (value) → integer – return number of occurrences of value

(list) **.extend()**  
L.extend(iterable) – extend list by appending elements from the iterable

(list) **.index** (value[, start[, stop]]) → integer – return first index of value.  
Raises ValueError if the value is not present.

(list) **.insert()**  
L.insert(index, object) – insert object before index

(list) **.pop** ([index]) → item – remove and return item at index (default last).  
Raises IndexError if list is empty or index is out of range.

(list) **.remove()**  
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

(list) **.reverse()**  
L.reverse() – reverse *IN PLACE*

(list) **.sort()**  
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

(list) **.\_\_add\_\_()**  
x.\_\_add\_\_(y) <==> x+y

(list) **.\_\_contains\_\_()**  
x.\_\_contains\_\_(y) <==> y in x

(list) **.\_\_delitem\_\_()**  
x.\_\_delitem\_\_(y) <==> del x[y]

(list) **.\_\_delslice\_\_()**  
x.\_\_delslice\_\_(i, j) <==> del x[i:j]  
Use of negative indices is not supported.

(list) **.\_\_eq\_\_()**  
x.\_\_eq\_\_(y) <==> x==y

(list) **.\_\_ge\_\_()**  
x.\_\_ge\_\_(y) <==> x>=y

(list) **.\_\_getitem\_\_()**  
x.\_\_getitem\_\_(y) <==> x[y]

(list) **.\_\_getslice\_\_()**  
x.\_\_getslice\_\_(i, j) <==> x[i:j]  
Use of negative indices is not supported.

(list) **.\_\_gt\_\_()**  
x.\_\_gt\_\_(y) <==> x>y

(list) **.\_\_iadd\_\_()**  
x.\_\_iadd\_\_(y) <==> x+=y

(list) **.\_\_imul\_\_()**  
x.\_\_imul\_\_(y) <==> x\*=y

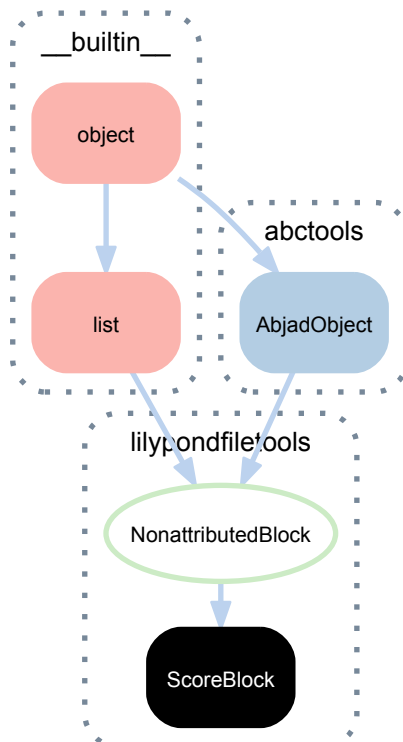
(list) **.\_\_iter\_\_()** <==> iter(x)

```

(list).__le__()
    x.__le__(y) <==> x<=y
(list).__len__() <==> len(x)
(list).__lt__()
    x.__lt__(y) <==> x<y
(list).__mul__()
    x.__mul__(n) <==> x*n
(list).__ne__()
    x.__ne__(y) <==> x!=y
(AttributedBlock).__repr__()
(list).__reversed__()
    L.__reversed__() – return a reverse iterator over the list
(list).__rmul__()
    x.__rmul__(n) <==> n*x
(list).__setitem__()
    x.__setitem__(i, y) <==> x[i]=y
(list).__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y
    Use of negative indices is not supported.

```

### 12.2.14 lilypondfiletools.ScoreBlock



**class** lilypondfiletools.**ScoreBlock**  
 Abjad model of LilyPond input file score block:

```
>>> score_block = lilypondfiletools.ScoreBlock()
```

```
>>> score_block
ScoreBlock()
```

```
>>> score_block.append(Staff([]))
>>> f(score_block)
\score {
  \new Staff {
  }
}
```

ScoreBlocks does not format when empty, as this generates a parser error in LilyPond:

```
>>> score_block = lilypondfiletools.ScoreBlock()
>>> score_block.lilypond_format == ''
True
```

Returns score block.

## Bases

- `lilypondfiletools.NonattributedBlock`
- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(NonattributedBlock).lilypond_format`

## Read/write properties

`(NonattributedBlock).is_formatted_when_empty`

## Methods

- `(list).append()`  
L.append(object) – append object to end
- `(list).count(value)` → integer – return number of occurrences of value
- `(list).extend()`  
L.extend(iterable) – extend list by appending elements from the iterable
- `(list).index(value[, start[, stop]])` → integer – return first index of value.  
Raises `ValueError` if the value is not present.
- `(list).insert()`  
L.insert(index, object) – insert object before index
- `(list).pop([index])` → item – remove and return item at index (default last).  
Raises `IndexError` if list is empty or index is out of range.
- `(list).remove()`  
L.remove(value) – remove first occurrence of value. Raises `ValueError` if the value is not present.
- `(list).reverse()`  
L.reverse() – reverse *IN PLACE*
- `(list).sort()`  
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

## Special methods

```
(list) .__add__()
x.__add__(y) <==> x+y
```

```
(list) .__contains__()
x.__contains__(y) <==> y in x
```

```
(list) .__delitem__()
x.__delitem__(y) <==> del x[y]
```

```
(list) .__delslice__()
x.__delslice__(i, j) <==> del x[i:j]
```

Use of negative indices is not supported.

```
(list) .__eq__()
x.__eq__(y) <==> x==y
```

```
(list) .__ge__()
x.__ge__(y) <==> x>=y
```

```
(list) .__getitem__()
x.__getitem__(y) <==> x[y]
```

```
(list) .__getslice__()
x.__getslice__(i, j) <==> x[i:j]
```

Use of negative indices is not supported.

```
(list) .__gt__()
x.__gt__(y) <==> x>y
```

```
(list) .__iadd__()
x.__iadd__(y) <==> x+=y
```

```
(list) .__imul__()
x.__imul__(y) <==> x*=y
```

```
(list) .__iter__() <==> iter(x)
```

```
(list) .__le__()
x.__le__(y) <==> x<=y
```

```
(list) .__len__() <==> len(x)
```

```
(list) .__lt__()
x.__lt__(y) <==> x<y
```

```
(list) .__mul__()
x.__mul__(n) <==> x*n
```

```
(list) .__ne__()
x.__ne__(y) <==> x!=y
```

```
(NonattributedBlock) .__repr__()
```

```
(list) .__reversed__()
L.__reversed__() – return a reverse iterator over the list
```

```
(list) .__rmul__()
x.__rmul__(n) <==> n*x
```

```
(list) .__setitem__()
x.__setitem__(i, y) <==> x[i]=y
```

```
(list) .__setslice__()
x.__setslice__(i, j, y) <==> x[i:j]=y
```

Use of negative indices is not supported.

## 12.3 Functions

### 12.3.1 `lilypondfiletools.make_basic_lilypond_file`

`lilypondfiletools.make_basic_lilypond_file` (*music=None*)

Make basic LilyPond file with *music*:

```
>>> score = Score([Staff("c'8 d'8 e'8 f'8")])
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
>>> lilypond_file.header_block.composer = markuptools.Markup('Josquin')
>>> lilypond_file.layout_block.indent = 0
>>> lilypond_file.paper_block.top_margin = 15
>>> lilypond_file.paper_block.left_margin = 15
```

Equip LilyPond file with header, layout and paper blocks.

Returns LilyPond file.

### 12.3.2 `lilypondfiletools.make_floating_time_signature_lilypond_file`

`lilypondfiletools.make_floating_time_signature_lilypond_file` (*music=None*)

Make floating time signature LilyPond file from *music*.

Function creates a basic LilyPond file.

Function then applies many layout settings.

[View source here](#) for the complete inventory of settings applied.

Returns LilyPond file object.

### 12.3.3 `lilypondfiletools.make_time_signature_context_block`

`lilypondfiletools.make_time_signature_context_block` (*font\_size=3, mini-*  
*mum\_distance=12,*  
*padding=4*)

Make time signature context block:

```
>>> context_block = lilypondfiletools.make_time_signature_context_block()
```

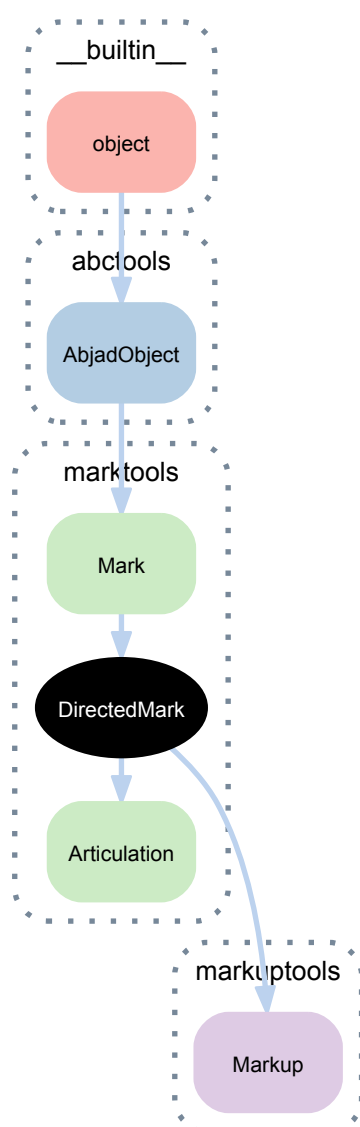
Returns context block.



# MARKTOOLS

## 13.1 Abstract classes

### 13.1.1 marktools.DirectedMark



```
class marktools.DirectedMark (*args, **kwargs)
```

Abstract base class for Marks which possess a vertical, typographic direction, i.e. above or below the staff.

## Bases

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

- (`Mark`) **.start\_component**  
Start component of mark.  
  
Returns component or none.
- (`Mark`) **.storage\_format**  
Storage format of mark.  
  
Returns string.

## Read/write properties

`DirectedMark.direction`

## Methods

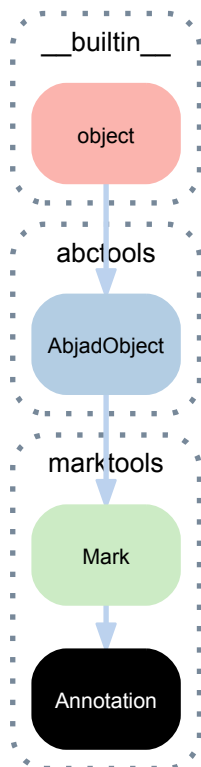
- (`Mark`) **.attach** (*start\_component*)  
Attaches mark to *start\_component*.  
  
Returns mark.
- (`Mark`) **.detach** ()  
Detaches mark from start component.  
  
Returns mark.

## Special methods

- (`Mark`) **.\_\_call\_\_** (\**args*)  
Detaches mark from component when called with no arguments.  
  
Attaches mark to component when called with one argument.  
  
Returns self.
- (`Mark`) **.\_\_copy\_\_** (\**args*)  
Copies mark.  
  
Returns new mark.
- (`Mark`) **.\_\_eq\_\_** (*expr*)  
True when *expr* is the same type as self. Otherwise false.  
  
Returns boolean.
- (`AbjadObject`) **.\_\_ne\_\_** (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
  
Returns boolean.
- (`Mark`) **.\_\_repr\_\_** ()  
Interpreter representation of mark.  
  
Returns string.

## 13.2 Concrete classes

### 13.2.1 marktools.Annotation



**class** `marktools.Annotation(*args)`  
 A user-defined annotation.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> pitch = pitchtools.NamedPitch('ds')
>>> marktools.Annotation('special pitch', pitch)(staff[0])
Annotation('special pitch', NamedPitch('ds'))(c'8)
```

Annotations contribute no formatting.

Annotations implement `__slots__`.

#### Bases

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(Mark).start_component`  
 Start component of mark.

Returns component or none.

`(Mark).storage_format`  
 Storage format of mark.

Returns string.

## Read/write properties

### Annotation.name

Get name of annotation:

```
>>> pitch = pitchtools.NamedPitch('ds')
>>> annotation = marktools.Annotation('special_pitch', pitch)
>>> annotation.name
'special_pitch'
```

Set name of annotation:

```
>>> annotation.name = 'revised special pitch'
>>> annotation.name
'revised special pitch'
```

Set string.

### Annotation.value

Get value of annotation:

```
>>> annotation.value
NamedPitch('ds')
```

Set value of annotation:

```
>>> annotation.value = pitchtools.NamedPitch('e')
>>> annotation.value
NamedPitch('e')
```

Set arbitrary object.

## Methods

(Mark) **.attach**(*start\_component*)

Attaches mark to *start\_component*.

Returns mark.

(Mark) **.detach**()

Detaches mark from start component.

Returns mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

Annotation **.\_\_copy\_\_**(\*args)

Annotation **.\_\_eq\_\_**(arg)

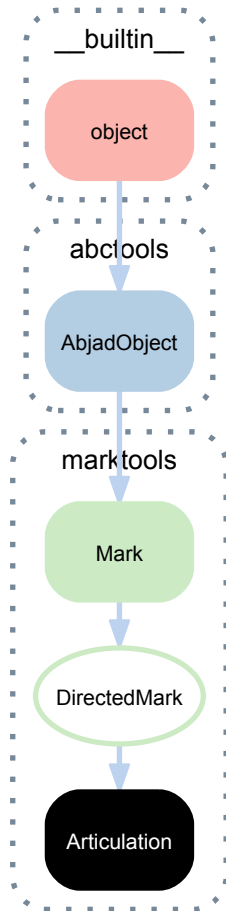
(AbjadObject) **.\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Mark).**\_\_repr\_\_**()  
 Interpreter representation of mark.  
 Returns string.

### 13.2.2 marktools.Articulation



**class** marktools.**Articulation**(\*args)  
 A musical articulation.

Initialize from articulation name:

```
>>> marktools.Articulation('staccato')
Articulation('staccato')
```

Initialize from articulation abbreviation:

```
>>> marktools.Articulation('.')
Articulation('.')
```

Initialize from other articulation:

```
>>> articulation = marktools.Articulation('staccato')
>>> marktools.Articulation(articulation)
Articulation('staccato')
```

Initialize with direction:

```
>>> marktools.Articulation('staccato', Up)
Articulation('staccato', Up)
```

Attach to note, rest or chord after initialization:

```
>>> note = Note("c'4")
```

```
>>> marktools.Articulation('staccato')(note)
Articulation('staccato')(c'4)
```

```
>>> f(note)
c'4 -\staccato
```

```
>>> show(note)
```



Articulations implement `__slots__`.

## Bases

- `marktools.DirectedMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`Articulation.lilypond_format`

LilyPond format string of articulation:

```
>>> articulation = marktools.Articulation('marcato', Up)
>>> articulation.lilypond_format
'^\marcato'
```

Returns string.

(`Mark`) `.start_component`

Start component of mark.

Returns component or none.

(`Mark`) `.storage_format`

Storage format of mark.

Returns string.

## Read/write properties

(`DirectedMark`) `.direction`

`Articulation.name`

Get name of articulation:

```
>>> articulation = marktools.Articulation('staccato', Up)
>>> articulation.name
'staccato'
```

Set name of articulation:

```
>>> articulation.name = 'marcato'
>>> articulation.name
'marcato'
```

Set string.

## Methods

(Mark) **.attach**(*start\_component*)  
Attaches mark to *start\_component*.  
Returns mark.

(Mark) **.detach**()  
Detaches mark from start component.  
Returns mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)  
Detaches mark from component when called with no arguments.  
Attaches mark to component when called with one argument.  
Returns self.

Articulation **.\_\_copy\_\_**(\*args)

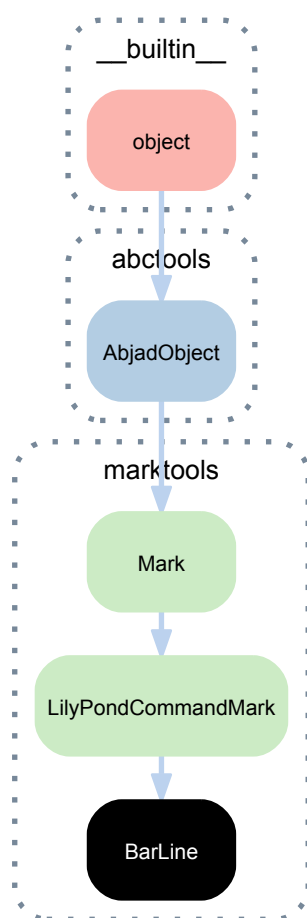
Articulation **.\_\_eq\_\_**(*expr*)

(AbjadObject) **.\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(Mark) **.\_\_repr\_\_**()  
Interpreter representation of mark.  
Returns string.

Articulation **.\_\_str\_\_**()

### 13.2.3 marktools.BarLine



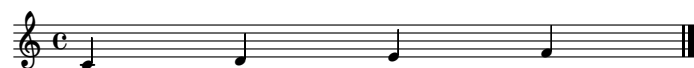
**class** `marktools.BarLine` (*bar\_line\_string='|', format\_slot='after'*)  
A bar line.

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
```

```
>>> bar_line = marktools.BarLine('|.') (staff[-1])
```

```
>>> bar_line
BarLine('|.') (f'4)
```

```
>>> show(staff)
```



Returns bar line.

#### Bases

- `marktools.LilyPondCommandMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`



## Read-only properties

`(LilyPondCommandMark).lilypond_format`  
LilyPond input format of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark(
...     'slurDotted')(note)
>>> lilypond_command.lilypond_format
'\\slurDotted'
```

Returns string.

`(Mark).start_component`  
Start component of mark.

Returns component or none.

`(Mark).storage_format`  
Storage format of mark.

Returns string.

## Read/write properties

`BarLine.bar_line_string`  
Get bar line string of bar line:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> bar_line = marktools.BarLine()(staff[-1])
>>> bar_line.bar_line_string
'|'
```

Set bar line string of bar line:

```
>>> bar_line.bar_line_string = '|.'
>>> bar_line.bar_line_string
'|.'
```

Set string.

`(LilyPondCommandMark).command_name`  
Get command name of LilyPond command mark:

```
>>> lilypond_command = marktools.LilyPondCommandMark(
...     'slurDotted')
>>> lilypond_command.command_name
'slurDotted'
```

Set command name of LilyPond command mark:

```
>>> lilypond_command.command_name = 'slurDashed'
>>> lilypond_command.command_name
'slurDashed'
```

Set string.

`(LilyPondCommandMark).format_slot`  
Get format slot of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark(
...     'break', 'after')
>>> lilypond_command.format_slot
'after'
```

Set format slot of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark(
...     'break', 'after')
>>> lilypond_command.format_slot = 'before'
>>> lilypond_command.format_slot
'before'
```

Set to 'before', 'after', 'opening', 'closing', 'right' or none.

## Methods

(Mark) **.attach**(*start\_component*)

Attaches mark to *start\_component*.

Returns mark.

(Mark) **.detach**()

Detaches mark from start component.

Returns mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

BarLine **.\_\_copy\_\_**(\*args)

(LilyPondCommandMark) **.\_\_eq\_\_**(arg)

(AbjadObject) **.\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

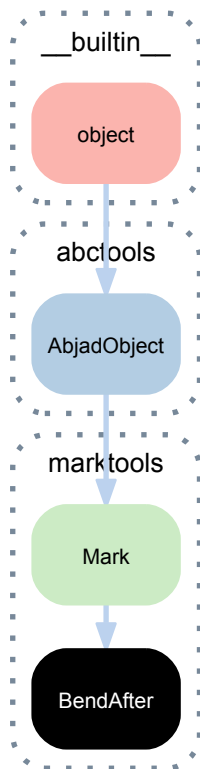
Returns boolean.

(Mark) **.\_\_repr\_\_**()

Interpreter representation of mark.

Returns string.

### 13.2.4 marktools.BendAfter



**class** marktools.**BendAfter** (\*args)  
*A fall or doit.*

```
>>> note = Note("c'4")
```

```
>>> marktools.BendAfter(-4)(note)
BendAfter(-4.0) (c'4)
```

```
>>> show(note)
```



BendAfter implements `__slots__`.

#### Bases

- marktools.Mark
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

BendAfter.**lilypond\_format**  
 LilyPond format string:

```
>>> bend = marktools.BendAfter(-4)
>>> bend.lilypond_format
"- \\bendAfter #' -4.0"
```

Returns string.

(Mark) **.start\_component**  
Start component of mark.  
Returns component or none.

(Mark) **.storage\_format**  
Storage format of mark.  
Returns string.

## Read/write properties

BendAfter **.bend\_amount**  
Get bend amount:

```
>>> bend = marktools.BendAfter(8)
>>> bend.bend_amount
8.0
```

Set bend amount:

```
>>> bend.bend_amount = -4
>>> bend.bend_amount
-4.0
```

Set float.

## Methods

(Mark) **.attach**(*start\_component*)  
Attaches mark to *start\_component*.  
Returns mark.

(Mark) **.detach**()  
Detaches mark from start component.  
Returns mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)  
Detaches mark from component when called with no arguments.  
Attaches mark to component when called with one argument.  
Returns self.

BendAfter **.\_\_copy\_\_**(\*args)

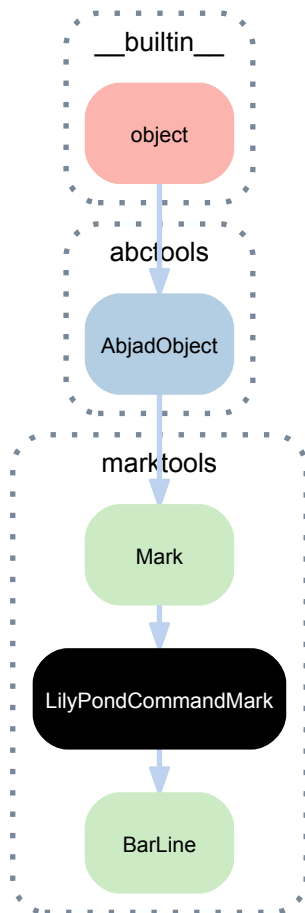
BendAfter **.\_\_eq\_\_**(*expr*)

(AbjadObject) **.\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(Mark) **.\_\_repr\_\_**()  
Interpreter representation of mark.  
Returns string.

BendAfter **.\_\_str\_\_**()

### 13.2.5 marktools.LilyPondCommandMark



**class** `marktools.LilyPondCommandMark(*args)`

A LilyPond command mark.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.select_leaves())
```

```
>>> lilypond_command = marktools.LilyPondCommandMark('slurDotted')
>>> lilypond_command.attach(staff[0])
LilyPondCommandMark('slurDotted')(c'8)
```

```
>>> show(staff)
```



Initialize LilyPond command marks from command name; or from command name with format slot; or from another LilyPond command mark; or from another LilyPond command mark with format slot.

LilyPond command marks implement `__slots__`.

#### Bases

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`LilyPondCommandMark.lilypond_format`

LilyPond input format of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark(
...     'slurDotted')(note)
>>> lilypond_command.lilypond_format
'\\slurDotted'
```

Returns string.

(Mark) **.start\_component**

Start component of mark.

Returns component or none.

(Mark) **.storage\_format**

Storage format of mark.

Returns string.

## Read/write properties

`LilyPondCommandMark.command_name`

Get command name of LilyPond command mark:

```
>>> lilypond_command = marktools.LilyPondCommandMark(
...     'slurDotted')
>>> lilypond_command.command_name
'slurDotted'
```

Set command name of LilyPond command mark:

```
>>> lilypond_command.command_name = 'slurDashed'
>>> lilypond_command.command_name
'slurDashed'
```

Set string.

`LilyPondCommandMark.format_slot`

Get format slot of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark(
...     'break', 'after')
>>> lilypond_command.format_slot
'after'
```

Set format slot of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark(
...     'break', 'after')
>>> lilypond_command.format_slot = 'before'
>>> lilypond_command.format_slot
'before'
```

Set to 'before', 'after', 'opening', 'closing', 'right' or none.

## Methods

(Mark) **.attach** (*start\_component*)

Attaches mark to *start\_component*.

Returns mark.

(Mark).**detach**()  
 Detaches mark from start component.  
 Returns mark.

### Special methods

(Mark).**\_\_call\_\_**(\*args)  
 Detaches mark from component when called with no arguments.  
 Attaches mark to component when called with one argument.  
 Returns self.

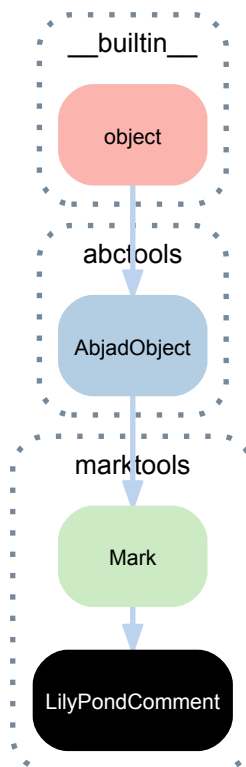
LilyPondCommandMark.**\_\_copy\_\_**(\*args)

LilyPondCommandMark.**\_\_eq\_\_**(arg)

(AbjadObject).**\_\_ne\_\_**(expr)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Mark).**\_\_repr\_\_**()  
 Interpreter representation of mark.  
 Returns string.

### 13.2.6 marktools.LilyPondComment



**class** marktools.**LilyPondComment**(\*args)  
 A user-defined LilyPond comment.

```
>>> note = Note("c'4")
```

```
>>> marktools.LilyPondComment('this is a comment')(note)
LilyPondComment('this is a comment')(c'4)
```

Initialize LilyPond comment from contents string; or contents string and format slot; or from other LilyPond comment; or from other LilyPond comment and format slot.

LilyPond comments implement `__slots__`.

## Bases

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`LilyPondComment.lilypond_format`

LilyPond input format of comment:

```
>>> comment = marktools.LilyPondComment('this is a comment.')
>>> comment.lilypond_format
'% this is a comment.'
```

Returns string.

(Mark) `.start_component`

Start component of mark.

Returns component or none.

(Mark) `.storage_format`

Storage format of mark.

Returns string.

## Read/write properties

`LilyPondComment.contents_string`

Get contents string of comment:

```
>>> comment = \
...     marktools.LilyPondComment('comment contents string')
>>> comment.contents_string
'comment contents string'
```

Set contents string of comment:

```
>>> comment.contents_string = 'new comment contents string'
>>> comment.contents_string
'new comment contents string'
```

Set string.

`LilyPondComment.format_slot`

Get format slot of LilyPond comment:

```
>>> note = Note("c'4")
>>> lilypond_comment = marktools.LilyPondComment('comment')
>>> lilypond_comment.format_slot
'before'
```

Set format slot of LilyPond comment:



```
>>> note = Note("c'4")
>>> lilypond_comment = marktools.LilyPondComment('comment')
>>> lilypond_comment.format_slot = 'after'
>>> lilypond_comment.format_slot
'after'
```

Set to 'before', 'after', 'opening', 'closing', 'right' or none.

## Methods

(Mark) **.attach** (*start\_component*)  
 Attaches mark to *start\_component*.  
 Returns mark.

(Mark) **.detach** ()  
 Detaches mark from start component.  
 Returns mark.

## Special methods

(Mark) **.\_\_call\_\_** (\*args)  
 Detaches mark from component when called with no arguments.  
 Attaches mark to component when called with one argument.  
 Returns self.

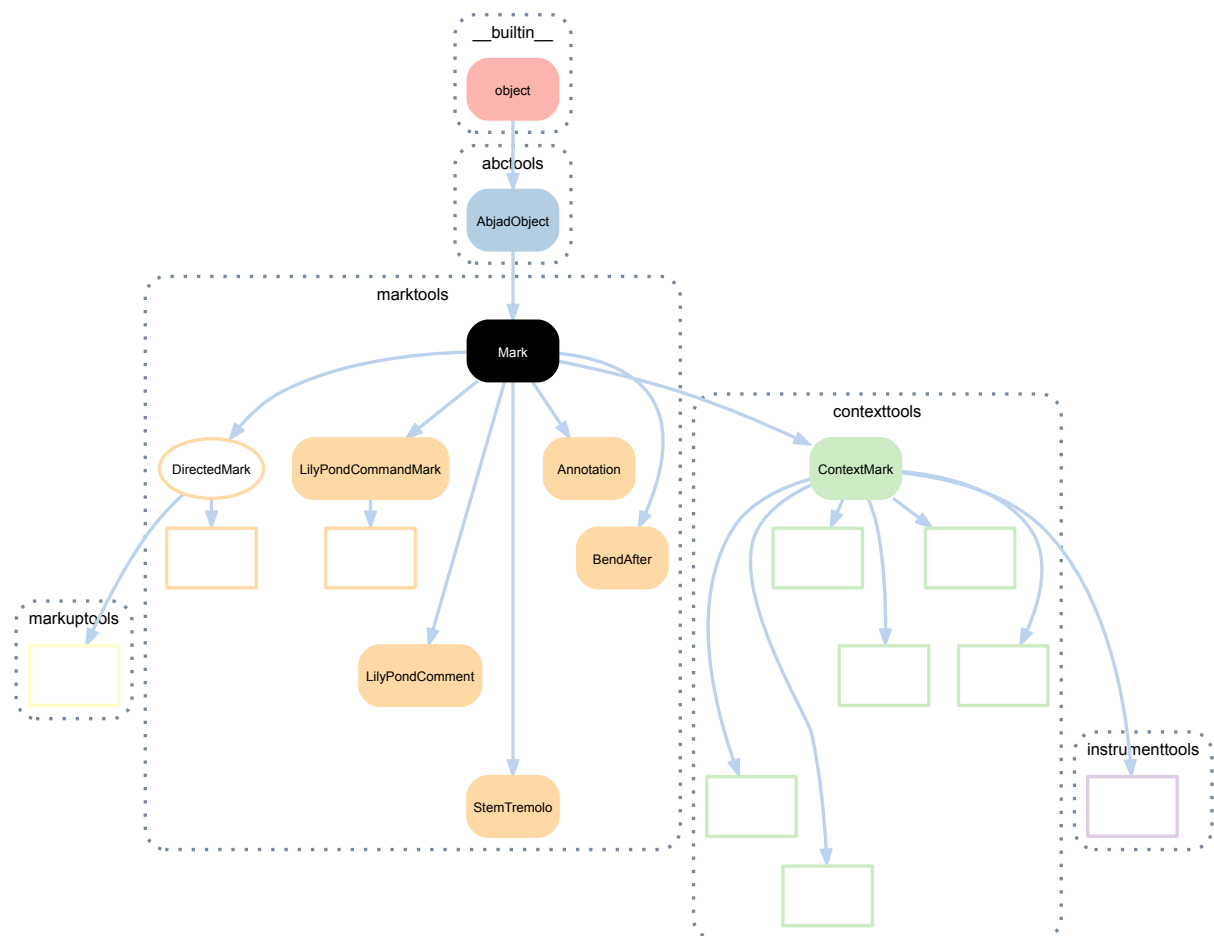
LilyPondComment **.\_\_copy\_\_** (\*args)

LilyPondComment **.\_\_eq\_\_** (arg)

(AbjadObject) **.\_\_ne\_\_** (expr)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Mark) **.\_\_repr\_\_** ()  
 Interpreter representation of mark.  
 Returns string.

## 13.2.7 marktools.Mark



**class** marktools.**Mark** (\*args)  
 Abstract base class from which concrete marks inherit.

### Bases

- abctools.AbjadObject
- \_\_builtin\_\_.object

### Read-only properties

**Mark.start\_component**  
 Start component of mark.  
 Returns component or none.

**Mark.storage\_format**  
 Storage format of mark.  
 Returns string.

### Methods

**Mark.attach** (start\_component)  
 Attaches mark to start\_component.  
 Returns mark.

`Mark.detach()`  
 Detaches mark from start component.  
 Returns mark.

### Special methods

`Mark.__call__(*args)`  
 Detaches mark from component when called with no arguments.  
 Attaches mark to component when called with one argument.  
 Returns self.

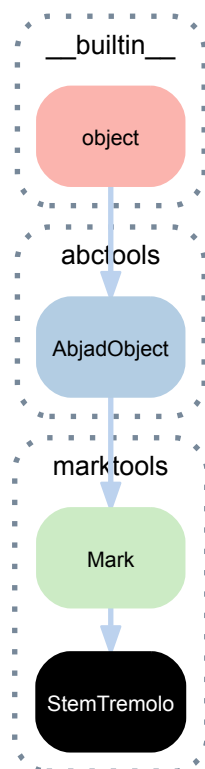
`Mark.__copy__(*args)`  
 Copies mark.  
 Returns new mark.

`Mark.__eq__(expr)`  
 True when *expr* is the same type as self. Otherwise false.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`Mark.__repr__()`  
 Interpreter representation of mark.  
 Returns string.

### 13.2.8 marktools.StemTremolo



**class** `marktools.StemTremolo(*args)`  
 A stem tremolo.

```
>>> note = Note("c'4")
```

```
>>> marktools.StemTremolo(16)(note)
StemTremolo(16)(c'4)
```

```
>>> show(note)
```



Stem tremolos implement `__slots__`.

## Bases

- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`StemTremolo.lilypond_format`  
 LilyPond format string:

```
>>> stem_tremolo = marktools.StemTremolo(16)
>>> stem_tremolo.lilypond_format
':16'
```

Returns string.

(`Mark`) **.start\_component**  
 Start component of mark.

Returns component or none.

(`Mark`) **.storage\_format**  
 Storage format of mark.

Returns string.

## Read/write properties

`StemTremolo.tremolo_flags`  
 Get tremolo flags:

```
>>> stem_tremolo = marktools.StemTremolo(16)
>>> stem_tremolo.tremolo_flags
16
```

Set tremolo flags:

```
>>> stem_tremolo.tremolo_flags = 32
>>> stem_tremolo.tremolo_flags
32
```

Set integer.

## Methods

(Mark) **.attach**(*start\_component*)  
Attaches mark to *start\_component*.

Returns mark.

(Mark) **.detach**()  
Detaches mark from start component.

Returns mark.

## Special methods

(Mark) **.\_\_call\_\_**(\*args)  
Detaches mark from component when called with no arguments.  
Attaches mark to component when called with one argument.  
Returns self.

StemTremolo **.\_\_copy\_\_**(\*args)  
Copy stem tremolo:

```
>>> import copy
>>> stem_tremolo_1 = marktools.StemTremolo(16)
>>> stem_tremolo_2 = copy.copy(stem_tremolo_1)
```

```
>>> stem_tremolo_1 == stem_tremolo_2
True
```

```
>>> stem_tremolo_1 is not stem_tremolo_2
True
```

Returns new stem tremolo.

StemTremolo **.\_\_eq\_\_**(*expr*)  
True when *expr* is a stem tremolo with equal tremolo flags: Otherwise false:

```
>>> stem_tremolo_1 = marktools.StemTremolo(16)
>>> stem_tremolo_2 = marktools.StemTremolo(16)
>>> stem_tremolo_3 = marktools.StemTremolo(32)
```

```
>>> stem_tremolo_1 == stem_tremolo_1
True
>>> stem_tremolo_1 == stem_tremolo_2
True
>>> stem_tremolo_1 == stem_tremolo_3
False
>>> stem_tremolo_2 == stem_tremolo_1
True
>>> stem_tremolo_2 == stem_tremolo_2
True
>>> stem_tremolo_2 == stem_tremolo_3
False
>>> stem_tremolo_3 == stem_tremolo_1
False
>>> stem_tremolo_3 == stem_tremolo_2
False
>>> stem_tremolo_3 == stem_tremolo_3
True
```

Returns boolean.

(AbjadObject) **.\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Mark).\_\_repr\_\_()

Interpreter representation of mark.

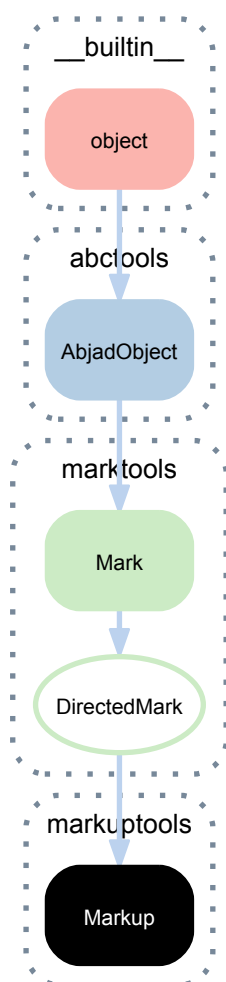
Returns string.

StemTremolo.\_\_str\_\_()

# MARKUPTOOLS

## 14.1 Concrete classes

### 14.1.1 markuptools.Markup



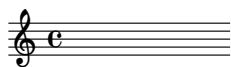
**class** markuptools.**Markup** (*argument*, *direction=None*, *markup\_name=None*)  
Abjad model of LilyPond markup.

Initialize from string:

```
>>> markup = markuptools.Markup(r'\bold { "This is markup text." }')
```

```
>>> markup
Markup((MarkupCommand('bold', ['This is markup text.']),))
```

```
>>> show(markup)
```



Initialize any markup from existing markup:

```
>>> markup_1 = markuptools.Markup('foo', direction=Up)
>>> markup_2 = markuptools.Markup(markup_1, direction=Down)
```

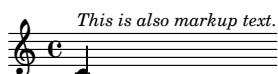
Attach markup to score components by calling them on the component:

```
>>> note = Note("c'4")
```

```
>>> markup = markuptools.Markup(
...     r'\italic { "This is also markup text." }', direction=Up)
```

```
>>> markup(note)
Markup((MarkupCommand('italic', ['This is also markup text.']),), direction=Up)(c'4)
```

```
>>> show(note)
```



Set *direction* to Up, Down, 'neutral', '^', '\_', '-' or None.

Markup objects are immutable.

Returns markup instance.

## Bases

- `marktools.DirectedMark`
- `marktools.Mark`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

Markup.**contents**

Tuple of contents of markup:

```
>>> markup = \
...     markuptools.Markup(r'\bold { "This is markup text." }')
>>> markup.contents
(MarkupCommand('bold', ['This is markup text.']),)
```

Returns string

Markup.**indented\_lilypond\_format**

Indented LilyPond format of markup:

```
>>> markup = \
...     markuptools.Markup(r'\bold { "This is markup text." }')
>>> print markup.indented_lilypond_format
\markup {
  \bold {
    "This is markup text."
```



```
}
}
```

Returns string.

`Markup.lilypond_format`

LilyPond format of markup:

```
>>> markup = \
...     markuptools.Markup(r'\bold { "This is markup text." }')
>>> markup.lilypond_format
'\markup { \bold { "This is markup text." } }'
```

Returns string.

`Markup.markup_name`

Name of markup:

```
>>> markup = markuptools.Markup(
...     r'\bold { allegro ma non troppo }',
...     markup_name='non troppo')
```

```
>>> markup.markup_name
'non troppo'
```

Returns string or none.

`(Mark).start_component`

Start component of mark.

Returns component or none.

`(Mark).storage_format`

Storage format of mark.

Returns string.

## Read/write properties

`(DirectedMark).direction`

## Methods

`(Mark).attach(start_component)`

Attaches mark to *start\_component*.

Returns mark.

`(Mark).detach()`

Detaches mark from start component.

Returns mark.

## Special methods

`(Mark).__call__(*args)`

Detaches mark from component when called with no arguments.

Attaches mark to component when called with one argument.

Returns self.

`Markup.__copy__(*args)`

`Markup.__eq__(expr)`

```
Markup.__hash__()
```

Markup.\_\_ne\_\_(*expr*)

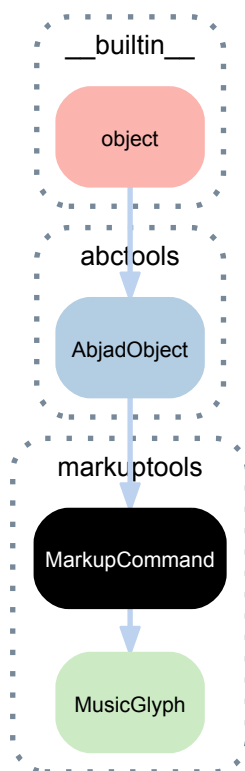
(Markup).\_\_repr\_\_()

Interpreter representation of mark.

Returns string.

```
Markup.__str__()
```

## 14.1.2 markuptools.MarkupCommand



**class** markuptools.**MarkupCommand**(*command*, \**args*)

Abjad model of a LilyPond markup command:

```
>>> circle = markuptools.MarkupCommand('draw-circle', 2.5, 0.1, False)
>>> square = markuptools.MarkupCommand('rounded-box', 'hello?')
>>> line = markuptools.MarkupCommand('line', [square, 'wow!'])
>>> rotate = markuptools.MarkupCommand('rotate', 60, line)
>>> combine = markuptools.MarkupCommand('combine', rotate, circle)
```

```
>>> print combine
\combine \rotate #60 \line { \rounded-box hello? wow! } \draw-circle #2.5 #0.1 ##f
```

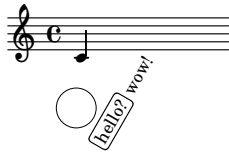
Insert a markup command in markup in order to attach it to score components:

```
>>> note = Note("c'4")
```

```
>>> markup = markuptools.Markup(combine)
```

```
>>> markup = markup(note)
```

```
>>> show(note)
```



Markup commands are immutable.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`MarkupCommand.args`

Tuple of markup command arguments.

`MarkupCommand.command`

String of markup command command-name.

`MarkupCommand.lilypond_format`

Format of markup command:

```
>>> markup_command = markuptools.MarkupCommand(
...     'draw-circle', 2.5, 0.1, False)
>>> markup_command.lilypond_format
'\\draw-circle #2.5 #0.1 ##f'
```

Returns string.

`MarkupCommand.storage_format`

Storage format of markup command.

Returns string.

## Special methods

`MarkupCommand.__eq__(expr)`

`(AbjadObject).__ne__(expr)`

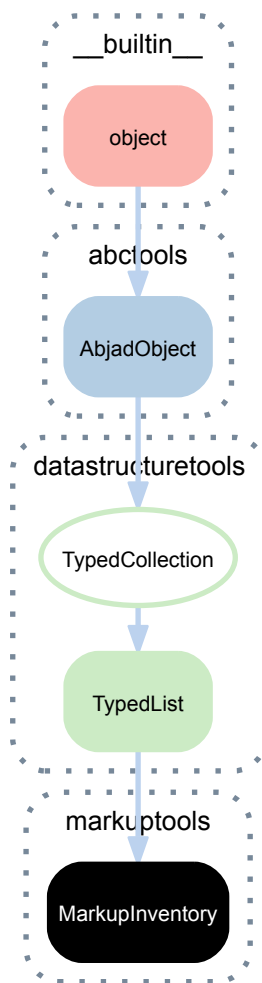
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`MarkupCommand.__repr__()`

`MarkupCommand.__str__()`

### 14.1.3 markuptools.MarkupInventory



**class** `markuptools.MarkupInventory` (*tokens=None, item\_class=None, name=None*)  
 Abjad model of an ordered list of markup:

```
>>> inventory = markuptools.MarkupInventory(['foo', 'bar'])
```

```
>>> inventory
MarkupInventory([Markup(('foo',)), Markup(('bar',))])
```

Markup inventories implement the list interface and are mutable.

#### Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(TypedCollection).item_class`  
 Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedList).append(token)`  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

`(TypedList).count(token)`  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)
3
```

Returns count.

`(TypedList).extend(tokens)`  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

`(TypedList).index(token)`  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('c'qf', "as'", 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

`(TypedList).insert(i, token)`  
Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['1', 2, 4.3])
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

(TypedList) **.pop** (*i=-1*)

Aliases list.pop().

(TypedList) **.remove** (*token*)

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['0', 1.0, 2, 3.14159])
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()

Aliases list.reverse().

(TypedList) **.sort** (*cmp=None, key=None, reverse=False*)

Aliases list.sort().

## Special methods

(TypedCollection) **.\_\_contains\_\_** (*token*)

(TypedList) **.\_\_delitem\_\_** (*i*)

Aliases list.\_\_delitem\_\_().

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedList) **.\_\_getitem\_\_** (*i*)

Aliases list.\_\_getitem\_\_().

(TypedList) **.\_\_iadd\_\_** (*expr*)

Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
        'ppp',
        target_context=stafftools.Staff
    ),
```

```

contexttools.DynamicMark (
    'p',
    target_context=stafftools.Staff
),
contexttools.DynamicMark (
    'mp',
    target_context=stafftools.Staff
),
contexttools.DynamicMark (
    'mf',
    target_context=stafftools.Staff
),
contexttools.DynamicMark (
    'fff',
    target_context=stafftools.Staff
)
],
item_class=contexttools.DynamicMark
)

```

Returns collection.

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

(TypedCollection).**\_\_ne\_\_**(*expr*)

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList).**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList).**\_\_setitem\_\_**(*i*, *expr*)

Change tokens in *expr* to items and set:

```

>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch('gqs,')
],
item_class=pitchtools.NamedPitch
)

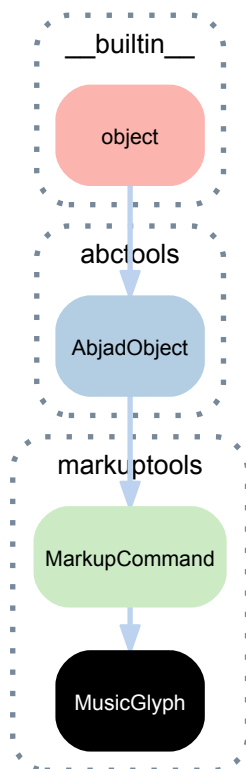
```

```

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch("f'"),
    pitchtools.NamedPitch("g'"),
    pitchtools.NamedPitch("a'"),
    pitchtools.NamedPitch("b'"),
    pitchtools.NamedPitch("c'")
],
item_class=pitchtools.NamedPitch
)

```

### 14.1.4 markuptools.MusicGlyph



**class** markuptools.**MusicGlyph**(*glyph\_name*)  
 Abjad model of a LilyPond musicglyph command:

```
>>> markuptools.MusicGlyph('accidentals.sharp')
MusicGlyph('accidentals.sharp')
>>> print _
\musicglyph #"accidentals.sharp"
```

Return *MusicGlyph* instance.

#### Bases

- markuptools.MarkupCommand
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

(MarkupCommand).**args**  
 Tuple of markup command arguments.

(MarkupCommand).**command**  
 String of markup command command-name.

(MarkupCommand).**lilypond\_format**  
 Format of markup command:

```
>>> markup_command = markuptools.MarkupCommand(
...     'draw-circle', 2.5, 0.1, False)
>>> markup_command.lilypond_format
'\draw-circle #2.5 #0.1 ##f'
```

Returns string.



(MarkupCommand).**storage\_format**  
 Storage format of markup command.  
 Returns string.

## Special methods

(MarkupCommand).**\_\_eq\_\_**(*expr*)  
 (AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

MusicGlyph.**\_\_repr\_\_**()  
 (MarkupCommand).**\_\_str\_\_**()

## 14.2 Functions

### 14.2.1 markuptools.combine\_markup\_commands

markuptools.**combine\_markup\_commands** (*\*commands*)  
 Combine MarkupCommand and/or string objects.

LilyPond's 'combine' markup command can only take two arguments, so in order to combine more than two stencils, a cascade of 'combine' commands must be employed. *combine\_markup\_commands* simplifies this process.

```
>>> markup_a = markuptools.MarkupCommand('draw-circle', 4, 0.4, False)
>>> markup_b = markuptools.MarkupCommand(
...     'filled-box',
...     schemetools.SchemePair(-4, 4),
...     schemetools.SchemePair(-0.5, 0.5), 1)
>>> markup_c = "some text"
```

```
>>> markup = markuptools.combine_markup_commands(markup_a, markup_b, markup_c)
>>> result = markup.lilypond_format
```

```
>>> print result
\combine \combine \draw-circle #4 #0.4 ##f
\filled-box #'(-4 . 4) #'(-0.5 . 0.5) #1 "some text"
```

Returns a markup command instance, or a string if that was the only argument.

### 14.2.2 markuptools.make\_big\_centered\_page\_number\_markup

markuptools.**make\_big\_centered\_page\_number\_markup** (*text=None*)  
 Make big centered page number markup:

```
>>> markup = markuptools.make_big_centered_page_number_markup()
```

```
>>> print markup.indented_lilypond_format
\markup {
  \fill-line
  {
    \bold
    \fontsize
    #3
    \concat
    {
      \on-the-fly
      #print-page-number-check-first
    }
  }
}
```

```
        \fromproperty
        #'page:page-number-string
    }
}
}
```

Returns markup.

### 14.2.3 `markuptools.make_blank_line_markup`

`markuptools.make_blank_line_markup()`

Make blank line markup:

```
>>> markup = markuptools.make_blank_line_markup()
```

```
>>> markup
Markup((MarkupCommand('fill-line', [' ']),))
```

Returns markup.

### 14.2.4 `markuptools.make_centered_title_markup`

`markuptools.make_centered_title_markup(title, font_name='Times', font_size=18, vspace_before=6, vspace_after=12)`

Make centered *title* markup:

```
>>> markup = markuptools.make_centered_title_markup('String Quartet')
```

```
>>> print markup.indented_lilypond_format
\markup {
  \override
    #'(font-name . "Times")
    \fontsize
      #18
      \column
        {
          \center-align
            {
              {
                \vspace
                  #6
                \line
                  {
                    "String Quartet"
                  }
                \vspace
                  #12
              }
            }
          }
        }
}
```

Returns markup.

### 14.2.5 `markuptools.make_vertically_adjusted_composer_markup`

`markuptools.make_vertically_adjusted_composer_markup(composer, font_name='Times', font_size=3, space_above=20, space_right=0)`

Make vertically adjusted *composer* markup:

```
>>> markup = markuptools.make_vertically_adjusted_composer_markup('Josquin Desprez')
```

```
>>> print markup.indented_lilypond_format
\markup {
  \override
    #'(font-name . "Times")
    {
      \hspace
        #0
      \raise
        #-20
        \fontsize
          #3
          "Josquin Desprez"
      \hspace
        #0
    }
}
```

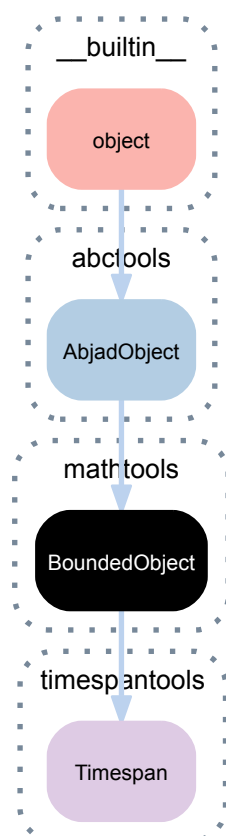
Returns markup.



# MATHTOOLS

## 15.1 Concrete classes

### 15.1.1 `mathtools.BoundedObject`



**class** `mathtools.BoundedObject`  
Bounded object mix-in.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`BoundedObject.is_closed`

`BoundedObject.is_half_closed`

`BoundedObject.is_half_open`

`BoundedObject.is_open`

### Read/write properties

`BoundedObject.is_left_closed`

`BoundedObject.is_left_open`

`BoundedObject.is_right_closed`

`BoundedObject.is_right_open`

### Special methods

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

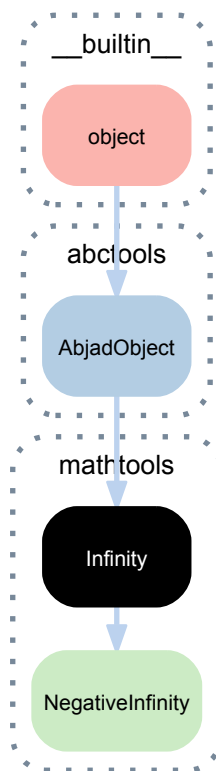
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 15.1.2 mathtools.Infinity



**class** `mathtools.Infinity`

Object-oriented infinity.

All numbers compare less than infinity:

```
>>> 9999999 < Infinity
True
```

```
>>> 2**38 < Infinity
True
```

Infinity compares equal to itself:

```
>>> Infinity == Infinity
True
```

Negative infinity compares less than infinity:

```
>>> NegativeInfinity < Infinity
True
```

Infinity is initialized at start-up and is available in the global Abjad namespace.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`Infinity.__eq__(expr)`

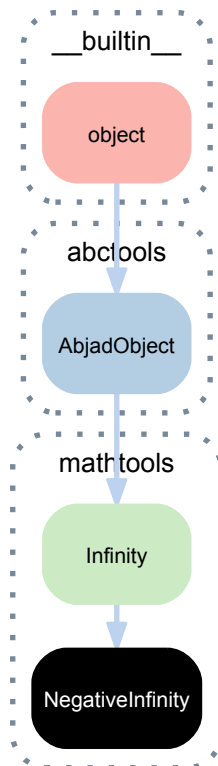
`Infinity.__ge__(expr)`

```

Infinity.__gt__(expr)
Infinity.__le__(expr)
Infinity.__lt__(expr)
(AbjadObject).__ne__(expr)
    True when ID of expr does not equal ID of Abjad object.
    Returns boolean.
Infinity.__repr__()

```

### 15.1.3 mathtools.NegativeInfinity



**class** `mathtools.NegativeInfinity`

Object-oriented negative infinity.

All numbers compare greater than negative infinity:

```

>>> NegativeInfinity < -9999999
True

```

Negative infinity compares equal to itself:

```

>>> NegativeInfinity == NegativeInfinity
True

```

Negative infinity compares less than infinity:

```

>>> NegativeInfinity < Infinity
True

```

Negative infinity is initialize at start-up and is available in the global Abjad namespace.

#### Bases

- `mathtools.Infinity`



- `abctools.AbjadObject`
- `__builtin__.object`

### Special methods

`(Infinity).__eq__(expr)`

`(Infinity).__ge__(expr)`

`(Infinity).__gt__(expr)`

`(Infinity).__le__(expr)`

`(Infinity).__lt__(expr)`

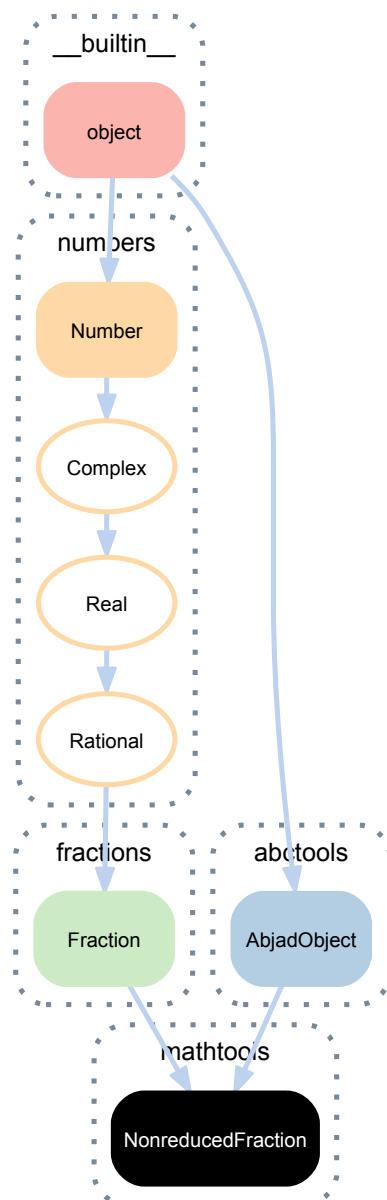
`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(Infinity).__repr__()`

## 15.1.4 mathtools.NonreducedFraction



**class** `mathtools.NonreducedFraction`

Initialize with an integer numerator and integer denominator:

```
>>> mathtools.NonreducedFraction(3, 6)
NonreducedFraction(3, 6)
```

Initialize with only an integer denominator:

```
>>> mathtools.NonreducedFraction(3)
NonreducedFraction(3, 1)
```

Initialize with an integer pair:

```
>>> mathtools.NonreducedFraction((3, 6))
NonreducedFraction(3, 6)
```

Initialize with an integer singleton:

```
>>> mathtools.NonreducedFraction((3,))
NonreducedFraction(3, 1)
```

Similar to built-in fraction except that numerator and denominator do not reduce.

Nonreduced fractions inherit from built-in fraction:

```
>>> isinstance(mathtools.NonreducedFraction(3, 6), Fraction)
True
```

Nonreduced fractions are numbers:

```
>>> import numbers
```

```
>>> isinstance(mathtools.NonreducedFraction(3, 6), numbers.Number)
True
```

Nonreduced fraction initialization requires more function calls than fraction initialization. But nonreduced fraction initialization is reasonably fast anyway:

```
>>> import fractions
>>> iotools.count_function_calls(
...     'fractions.Fraction(3, 6)', globals())
13
```

```
>>> iotools.count_function_calls(
...     'mathtools.NonreducedFraction(3, 6)', globals())
30
```

Nonreduced fractions are immutable.

## Bases

- `abctools.AbjadObject`
- `fractions.Fraction`
- `numbers.Rational`
- `numbers.Real`
- `numbers.Complex`
- `numbers.Number`
- `__builtin__.object`

## Read-only properties

`NonreducedFraction.denominator`

Denominator of nonreduced fraction.

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> fraction.denominator
3
```

Returns positive integer.

`NonreducedFraction.imag`

Nonreduced fractions have no imaginary part.

```
>>> fraction.imag
0
```

Returns zero.

`NonreducedFraction.numerator`

Numerator of nonreduced fraction.

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> fraction.numerator
-6
```

Returns integer.

`NonreducedFraction.pair`

Read only pair of nonreduced fraction numerator and denominator.

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> fraction.pair
(-6, 3)
```

Returns integer pair.

`NonreducedFraction.real`

Nonreduced fractions are their own real component.

```
>>> fraction.real
NonreducedFraction(-6, 3)
```

Returns nonreduced fraction.

`NonreducedFraction.storage_format`

Storage format of nonreduced fraction.

```
>>> print fraction.storage_format
mathtools.NonreducedFraction(-6, 3)
```

Returns string.

## Methods

`(Real).conjugate()`

Conjugate is a no-op for Reals.

`(Fraction).limit_denominator(max_denominator=1000000)`

Closest Fraction to self with denominator at most max\_denominator.

```
>>> Fraction('3.141592653589793').limit_denominator(10)
Fraction(22, 7)
>>> Fraction('3.141592653589793').limit_denominator(100)
Fraction(311, 99)
>>> Fraction(4321, 8765).limit_denominator(10000)
Fraction(4321, 8765)
```

`NonreducedFraction.multiply_with_cross_cancelation(multiplier)`

Multiplies nonreduced fraction by *expr* with cross-cancelation.

```
>>> fraction = mathtools.NonreducedFraction(4, 8)
```

```
>>> fraction.multiply_with_cross_cancelation((2, 3))
NonreducedFraction(4, 12)
```

```
>>> fraction.multiply_with_cross_cancelation((4, 1))
NonreducedFraction(4, 2)
```

```
>>> fraction.multiply_with_cross_cancelation((3, 5))
NonreducedFraction(12, 40)
```

```
>>> fraction.multiply_with_cross_cancelation((6, 5))
NonreducedFraction(12, 20)
```

```
>>> fraction = mathtools.NonreducedFraction(5, 6)
>>> fraction.multiply_with_cross_cancelation((6, 5))
NonreducedFraction(1, 1)
```

Returns nonreduced fraction.

`NonreducedFraction.multiply_with_numerator_preservation(multiplier)`  
 Multiplies nonreduced fraction by *multiplier* with numerator preservation where possible.

```
>>> fraction = mathtools.NonreducedFraction(9, 16)
```

```
>>> fraction.multiply_with_numerator_preservation((2, 3))
NonreducedFraction(9, 24)
```

```
>>> fraction.multiply_with_numerator_preservation((1, 2))
NonreducedFraction(9, 32)
```

```
>>> fraction.multiply_with_numerator_preservation((5, 6))
NonreducedFraction(45, 96)
```

```
>>> fraction = mathtools.NonreducedFraction(3, 8)
```

```
>>> fraction.multiply_with_numerator_preservation((2, 3))
NonreducedFraction(3, 12)
```

Returns nonreduced fraction.

`NonreducedFraction.multiply_without_reducing(expr)`  
 Multiplies nonreduced fraction by *expr* without reducing.

```
>>> fraction = mathtools.NonreducedFraction(3, 8)
```

```
>>> fraction.multiply_without_reducing((3, 3))
NonreducedFraction(9, 24)
```

```
>>> fraction = mathtools.NonreducedFraction(4, 8)
```

```
>>> fraction.multiply_without_reducing((4, 5))
NonreducedFraction(16, 40)
```

```
>>> fraction.multiply_without_reducing((3, 4))
NonreducedFraction(12, 32)
```

Returns nonreduced fraction.

`NonreducedFraction.reduce()`  
 Reduces nonreduced fraction.

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> fraction.reduce()
Fraction(-2, 1)
```

Returns fraction.

`NonreducedFraction.with_denominator(denominator)`  
 Returns new nonreduced fraction with integer *denominator*.

```
>>> mathtools.NonreducedFraction(3, 6).with_denominator(12)
NonreducedFraction(6, 12)
```

Returns nonreduced fraction.

`NonreducedFraction.with_multiple_of_denominator(denominator)`  
 Returns new nonreduced fraction with multiple of integer *denominator*.

```
>>> fraction = mathtools.NonreducedFraction(3, 6)
```

```
>>> fraction.with_multiple_of_denominator(5)
NonreducedFraction(5, 10)
```

Returns nonreduced fraction.

## Class methods

(Fraction).**from\_decimal**(*dec*)  
Converts a finite Decimal instance to a rational number, exactly.

(Fraction).**from\_float**(*f*)  
Converts a finite float to a rational number, exactly.  
Beware that Fraction.from\_float(0.3) != Fraction(3, 10).

## Special methods

NonreducedFraction.**\_\_abs\_\_**()  
Absolute value of nonreduced fraction.

```
>>> abs(mathtools.NonreducedFraction(-3, 3))
NonreducedFraction(3, 3)
```

Returns nonreduced fraction.

NonreducedFraction.**\_\_add\_\_**(*expr*)  
Adds *expr* to nonreduced fraction.

```
>>> mathtools.NonreducedFraction(3, 3) + 1
NonreducedFraction(6, 3)
```

Adding two nonreduced fractions is fairly fast:

```
>>> a = mathtools.NonreducedFraction(3, 6)
>>> b = mathtools.NonreducedFraction(3, 12)
>>> iotools.count_function_calls('a + b', globals())
67
```

Adding an integer is even faster:

```
>>> iotools.count_function_calls('a + 10', globals())
35
```

Returns nonreduced fraction.

(Real).**\_\_complex\_\_**()  
complex(self) == complex(float(self), 0)

(Fraction).**\_\_copy\_\_**()

(Fraction).**\_\_deepcopy\_\_**(*memo*)

NonreducedFraction.**\_\_div\_\_**(*expr*)  
Divides nonreduced fraction by *expr*.

```
>>> mathtools.NonreducedFraction(3, 3) / 1
NonreducedFraction(3, 3)
```

Returns nonreduced fraction.

(Real).**\_\_divmod\_\_**(*other*)  
divmod(self, other): The pair (self // other, self % other).

Sometimes this can be computed faster than the pair of operations.

NonreducedFraction.**\_\_eq\_\_**(*expr*)  
True when *expr* equals nonreduced fraction.

```
>>> mathtools.NonreducedFraction(3, 3) == 1
True
```

Returns boolean.

```
(Rational).__float__()
float(self) = self.numerator / self.denominator
```

It's important that this conversion use the integer's “true” division rather than casting one side to float before dividing so that ratios of huge integers convert without overflowing.

```
(Fraction).__floordiv__(a, b)
a // b
```

```
NonreducedFraction.__ge__(expr)
True when nonreduced fraction is greater than or equal to expr.
```

```
>>> mathtools.NonreducedFraction(3, 3) >= 1
True
```

Returns boolean.

```
NonreducedFraction.__gt__(expr)
True when nonreduced fraction is greater than expr.
```

```
>>> mathtools.NonreducedFraction(3, 3) > 1
False
```

Returns boolean.

```
(Fraction).__hash__()
hash(self)
```

Tricky because values that are exactly representable as a float must have the same hash as that float.

```
NonreducedFraction.__le__(expr)
True when nonreduced fraction is less than or equal to expr.
```

```
>>> mathtools.NonreducedFraction(3, 3) <= 1
True
```

Returns boolean.

```
NonreducedFraction.__lt__(expr)
True when nonreduced fraction is less than expr.
```

```
>>> mathtools.NonreducedFraction(3, 3) < 1
False
```

Returns boolean.

```
(Fraction).__mod__(a, b)
a % b
```

```
NonreducedFraction.__mul__(expr)
Multiplies nonreduced fraction by expr.
```

```
>>> mathtools.NonreducedFraction(3, 3) * 3
NonreducedFraction(9, 3)
```

Returns nonreduced fraction.

```
NonreducedFraction.__ne__(expr)
True when expr does not equal nonreduced fraction.
```

```
>>> mathtools.NonreducedFraction(3, 3) != 'foo'
True
```

Returns boolean.

`NonreducedFraction.__neg__()`  
Negates nonreduced fraction.

```
>>> -mathtools.NonreducedFraction(3, 3)
NonreducedFraction(-3, 3)
```

Returns nonreduced fraction.

`NonreducedFraction.__new__(*args)`

`(Fraction).__nonzero__(a)`  
`a != 0`

`(Fraction).__pos__(a)`  
+a: Coerces a subclass instance to Fraction

`NonreducedFraction.__pow__(expr)`  
Raises nonreduced fraction to *expr*.

```
>>> mathtools.NonreducedFraction(3, 6) ** -1
NonreducedFraction(6, 3)
```

Returns nonreduced fraction.

`NonreducedFraction.__radd__(expr)`  
Adds nonreduced fraction to *expr*.

```
>>> 1 + mathtools.NonreducedFraction(3, 3)
NonreducedFraction(6, 3)
```

Returns nonreduced fraction.

`NonreducedFraction.__rdiv__(expr)`  
Divides *expr* by nonreduced fraction.

```
>>> 1 / mathtools.NonreducedFraction(3, 3)
NonreducedFraction(3, 3)
```

Returns nonreduced fraction.

`(Real).__rdivmod__(other)`  
divmod(other, self): The pair (self // other, self % other).

Sometimes this can be computed faster than the pair of operations.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.

Returns string.

`(Fraction).__rfloordiv__(b, a)`  
`a // b`

`(Fraction).__rmod__(b, a)`  
`a % b`

`NonreducedFraction.__rmul__(expr)`  
Multiplies *expr* by nonreduced fraction.

```
>>> 3 * mathtools.NonreducedFraction(3, 3)
NonreducedFraction(9, 3)
```

Returns nonreduced fraction.

`(Fraction).__rpow__(b, a)`  
`a ** b`

`NonreducedFraction.__rsub__(expr)`  
Subtracts nonreduced fraction from *expr*.



```
>>> 1 - mathtools.NonreducedFraction(3, 3)
NonreducedFraction(0, 3)
```

Returns nonreduced fraction.

```
(Fraction).__rtruediv__(b, a)
a / b
```

```
NonreducedFraction.__str__()
String representation of nonreduced fraction.
```

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> str(fraction)
'-6/3'
```

Returns string.

```
NonreducedFraction.__sub__(expr)
Subtracts expr from nonreduced fraction.
```

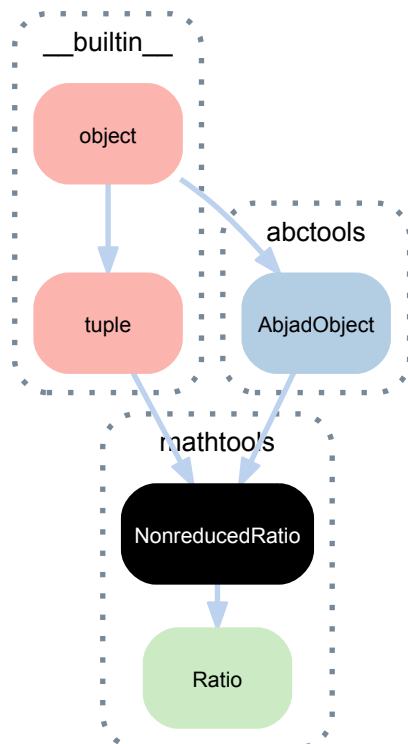
```
>>> mathtools.NonreducedFraction(3, 3) - 2
NonreducedFraction(-3, 3)
```

Returns nonreduced fraction.

```
(Fraction).__truediv__(a, b)
a / b
```

```
(Fraction).__trunc__(a)
trunc(a)
```

### 15.1.5 mathtools.NonreducedRatio



**class** `mathtools.NonreducedRatio`  
 Nonreduced ratio of one or more nonzero integers.  
 Initialize from one or more nonzero integers:

```
>>> mathtools.NonreducedRatio(2, 4, 2)
NonreducedRatio(2, 4, 2)
```

Or initialize from a tuple or list:

```
>>> ratio = mathtools.NonreducedRatio((2, 4, 2))
>>> ratio
NonreducedRatio(2, 4, 2)
```

Use a tuple to return ratio integers.

```
>>> tuple(ratio)
(2, 4, 2)
```

Nonreduced ratios are immutable.

## Bases

- `abctools.AbjadObject`
- `__builtin__.tuple`
- `__builtin__.object`

## Methods

`(tuple).count(value)` → integer – return number of occurrences of value

`(tuple).index(value[, start[, stop]])` → integer – return first index of value.

Raises `ValueError` if the value is not present.

## Special methods

`(tuple).__add__()`  
`x.__add__(y) <==> x+y`

`(tuple).__contains__()`  
`x.__contains__(y) <==> y in x`

`NonreducedRatio.__eq__(expr)`

`(tuple).__ge__()`  
`x.__ge__(y) <==> x>=y`

`(tuple).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(tuple).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(tuple).__gt__()`  
`x.__gt__(y) <==> x>y`

`(tuple).__hash__()` <==> `hash(x)`

`(tuple).__iter__()` <==> `iter(x)`

`(tuple).__le__()`  
`x.__le__(y) <==> x<=y`

`(tuple).__len__()` <==> `len(x)`

`(tuple).__lt__()`  
`x.__lt__(y) <==> x<y`

```
(tuple).__mul__()
x.__mul__(n) <==> x*n
```

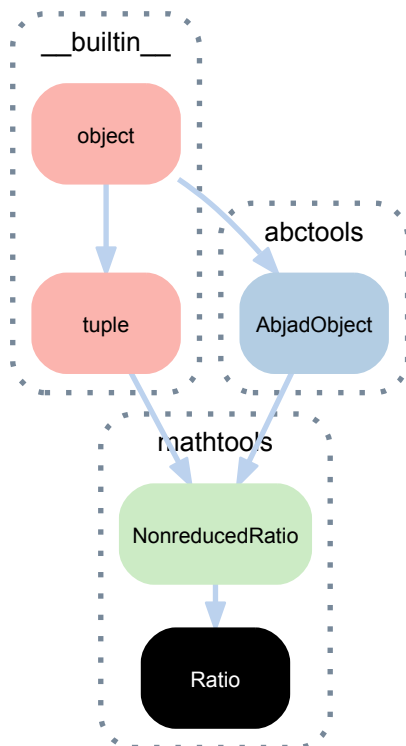
(AbjadObject).\_\_ne\_\_(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

```
NonreducedRatio.__new__(*args)
```

(AbjadObject).\_\_repr\_\_()  
Interpreter representation of Abjad object.  
Returns string.

```
(tuple).__rmul__()
x.__rmul__(n) <==> n*x
```

### 15.1.6 mathtools.Ratio



**class** `mathtools.Ratio`

Ratio of one or more nonzero integers.

Initialize from one or more nonzero integers:

```
>>> mathtools.Ratio(2, 4, 2)
Ratio(1, 2, 1)
```

Or initialize from a tuple or list:

```
>>> ratio = mathtools.Ratio((2, 4, 2))
>>> ratio
Ratio(1, 2, 1)
```

Use a tuple to return ratio integers.

```
>>> tuple(ratio)
(1, 2, 1)
```

Ratios are immutable.

## Bases

- `mathtools.NonreducedRatio`
- `abctools.AbjadObject`
- `__builtin__.tuple`
- `__builtin__.object`

## Methods

`(tuple).count(value)` → integer – return number of occurrences of value

`(tuple).index(value[, start[, stop]])` → integer – return first index of value.  
Raises `ValueError` if the value is not present.

## Special methods

`(tuple).__add__()`  
`x.__add__(y) <==> x+y`

`(tuple).__contains__()`  
`x.__contains__(y) <==> y in x`

`(NonreducedRatio).__eq__(expr)`

`(tuple).__ge__()`  
`x.__ge__(y) <==> x>=y`

`(tuple).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(tuple).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(tuple).__gt__()`  
`x.__gt__(y) <==> x>y`

`(tuple).__hash__()` <==> `hash(x)`

`(tuple).__iter__()` <==> `iter(x)`

`(tuple).__le__()`  
`x.__le__(y) <==> x<=y`

`(tuple).__len__()` <==> `len(x)`

`(tuple).__lt__()`  
`x.__lt__(y) <==> x<y`

`(tuple).__mul__()`  
`x.__mul__(n) <==> x*n`

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`Ratio.__new__(*args)`

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

```
(tuple).__rmul__()  
x.__rmul__(n) <==> n*x
```

## 15.2 Functions

### 15.2.1 `mathtools.are_relatively_prime`

`mathtools.are_relatively_prime` (*expr*)

True when *expr* is a sequence comprising zero or more numbers, all of which are relatively prime.

```
>>> mathtools.are_relatively_prime([13, 14, 15])  
True
```

Otherwise false:

```
>>> mathtools.are_relatively_prime([13, 14, 15, 16])  
False
```

Returns true when *expr* is an empty sequence:

```
>>> mathtools.are_relatively_prime([])  
True
```

Returns false when *expr* is nonsensical type:

```
>>> mathtools.are_relatively_prime('foo')  
False
```

Returns boolean.

### 15.2.2 `mathtools.arithmetic_mean`

`mathtools.arithmetic_mean` (*sequence*)

Arithmetic means of *sequence* as an exact integer.

```
>>> mathtools.arithmetic_mean([1, 2, 2, 20, 30])  
11
```

As a rational:

```
>>> mathtools.arithmetic_mean([1, 2, 20])  
Fraction(23, 3)
```

As a float:

```
>>> mathtools.arithmetic_mean([2, 2, 20.0])  
8.0
```

Returns number.

### 15.2.3 `mathtools.binomial_coefficient`

`mathtools.binomial_coefficient` (*n*, *k*)

Binomial coefficient of *n* choose *k*.

```
>>> for k in range(8):  
...     print k, '\t', mathtools.binomial_coefficient(8, k)  
...  
0 1  
1 8  
2 28  
3 56  
4 70
```

```
5 56
6 28
7 8
```

Returns positive integer.

### 15.2.4 `mathtools.cumulative_products`

`mathtools.cumulative_products(sequence)`  
Cumulative products of *sequence*.

```
>>> mathtools.cumulative_products([1, 2, 3, 4, 5, 6, 7, 8])
[1, 2, 6, 24, 120, 720, 5040, 40320]
```

```
>>> mathtools.cumulative_products([1, -2, 3, -4, 5, -6, 7, -8])
[1, -2, -6, 24, 120, -720, -5040, 40320]
```

Raises type error when *sequence* is neither list nor tuple.

Raises value error on empty *sequence*.

Returns list.

### 15.2.5 `mathtools.cumulative_signed_weights`

`mathtools.cumulative_signed_weights(sequence)`  
Cumulative signed weights of *sequence*.

```
>>> l = [1, -2, -3, 4, -5, -6, 7, -8, -9, 10]
>>> mathtools.cumulative_signed_weights(l)
[1, -3, -6, 10, -15, -21, 28, -36, -45, 55]
```

Raises type error when *sequence* is not a list.

Use `mathtools.cumulative_sums([abs(x) for x in l])` for cumulative (unsigned) weights

Returns list.

### 15.2.6 `mathtools.cumulative_sums`

`mathtools.cumulative_sums(sequence)`  
Cumulative sums of *sequence*.

```
>>> mathtools.cumulative_sums([1, 2, 3, 4, 5, 6, 7, 8])
[1, 3, 6, 10, 15, 21, 28, 36]
```

Raises type error when *sequence* is neither list nor tuple.

Raises value error on empty *sequence*.

Returns list.

### 15.2.7 `mathtools.cumulative_sums_zero`

`mathtools.cumulative_sums_zero(sequence)`  
Cumulative sums of *sequence* starting from 0.

```
>>> mathtools.cumulative_sums_zero([1, 2, 3, 4, 5, 6, 7, 8])
[0, 1, 3, 6, 10, 15, 21, 28, 36]
```

Returns `[0]` on empty *sequence*:

```
>>> mathtools.cumulative_sums_zero([])
[0]
```

Returns list.

### 15.2.8 mathtools.cumulative\_sums\_zero\_pairwise

`mathtools.cumulative_sums_zero_pairwise(sequence)`  
 Lists pairwise cumulative sums of *sequence* from 0.

```
>>> mathtools.cumulative_sums_zero_pairwise([1, 2, 3, 4, 5, 6])
[(0, 1), (1, 3), (3, 6), (6, 10), (10, 15), (15, 21)]
```

Returns list of pairs.

### 15.2.9 mathtools.difference\_series

`mathtools.difference_series(sequence)`  
 Difference series of *sequence*.

```
>>> mathtools.difference_series([1, 1, 2, 3, 5, 5, 6])
[0, 1, 1, 1, 2, 0, 1]
```

Returns list.

### 15.2.10 mathtools.divide\_number\_by\_ratio

`mathtools.divide_number_by_ratio(number, ratio)`  
 Divides integer by *ratio*.

```
>>> mathtools.divide_number_by_ratio(1, [1, 1, 3])
[Fraction(1, 5), Fraction(1, 5), Fraction(3, 5)]
```

Divides fraction by *ratio*:

```
>>> mathtools.divide_number_by_ratio(Fraction(1), [1, 1, 3])
[Fraction(1, 5), Fraction(1, 5), Fraction(3, 5)]
```

Divides float by ratio:

```
>>> mathtools.divide_number_by_ratio(1.0, [1, 1, 3])
[0.20000000000000001, 0.20000000000000001, 0.60000000000000009]
```

Raises type error on nonnumeric *number*.

Raises type error on noninteger in *ratio*.

Returns list of fractions or list of floats.

### 15.2.11 mathtools.divisors

`mathtools.divisors(n)`  
 Positive divisors of integer *n* in increasing order.

```
>>> mathtools.divisors(84)
[1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42, 84]
```

```
>>> for x in range(10, 20):
...     print x, mathtools.divisors(x)
...
10 [1, 2, 5, 10]
11 [1, 11]
```

```
12 [1, 2, 3, 4, 6, 12]
13 [1, 13]
14 [1, 2, 7, 14]
15 [1, 3, 5, 15]
16 [1, 2, 4, 8, 16]
17 [1, 17]
18 [1, 2, 3, 6, 9, 18]
19 [1, 19]
```

Allows nonpositive  $n$ :

```
>>> mathtools.divisors(-27)
[1, 3, 9, 27]
```

Performance is extremely fast:

```
>>> iotools.count_function_calls(
...     'mathtools.divisors(100000000)',
...     globals(),
...     )
50
```

Raises type error on noninteger  $n$ .

Raises not implemented error on 0.

Returns list of positive integers.

### 15.2.12 mathtools.factors

`mathtools.factors( $n$ )`

Integer factors of positive integer  $n$  in increasing order.

```
>>> mathtools.factors(84)
[1, 2, 2, 3, 7]
```

```
>>> for n in range(10, 20):
...     print n, mathtools.factors(n)
...
10 [1, 2, 5]
11 [1, 11]
12 [1, 2, 2, 3]
13 [1, 13]
14 [1, 2, 7]
15 [1, 3, 5]
16 [1, 2, 2, 2, 2]
17 [1, 17]
18 [1, 2, 3, 3]
19 [1, 19]
```

Raises type error on noninteger  $n$ .

Raises value error on nonpositive  $n$ .

Returns list of one or more positive integers.

### 15.2.13 mathtools.fraction\_to\_proper\_fraction

`mathtools.fraction_to_proper_fraction( $rational$ )`

Changes *rational* to proper fraction.

```
>>> mathtools.fraction_to_proper_fraction(Fraction(116, 8))
(14, Fraction(1, 2))
```

Returns pair.



### 15.2.14 `mathtools.get_shared_numeric_sign`

`mathtools.get_shared_numeric_sign(sequence)`

Gets shared numeric sign of elements in *sequence*.

Returns 1 when all *sequence* elements are positive:

```
>>> mathtools.get_shared_numeric_sign([1, 2, 3])
1
```

Returns -1 when all *sequence* elements are negative:

```
>>> mathtools.get_shared_numeric_sign([-1, -2, -3])
-1
```

Returns 0 on empty *sequence*:

```
>>> mathtools.get_shared_numeric_sign([])
0
```

Otherwise returns none:

```
>>> mathtools.get_shared_numeric_sign([1, 2, -3]) is None
True
```

Returns 1, -1, 0 or none.

### 15.2.15 `mathtools.greatest_common_divisor`

`mathtools.greatest_common_divisor(*integers)`

Calculates greatest common divisor of *integers*.

```
>>> mathtools.greatest_common_divisor(84, -94, -144)
2
```

Allows nonpositive input.

Raises type error on noninteger input.

Raises not implemented error when 0 is included in input.

Returns positive integer.

### 15.2.16 `mathtools.greatest_multiple_less_equal`

`mathtools.greatest_multiple_less_equal(m, n)`

Greatest integer multiple of *m* less than or equal to *n*.

```
>>> mathtools.greatest_multiple_less_equal(10, 47)
40
```

```
>>> for m in range(1, 10):
...     print m, mathtools.greatest_multiple_less_equal(m, 47)
...
1 47
2 46
3 45
4 44
5 45
6 42
7 42
8 40
9 45
```

```
>>> for n in range(10, 100, 10):
...     print mathtools.greatest_multiple_less_equal(7, n), n
...
7 10
14 20
28 30
35 40
49 50
56 60
70 70
77 80
84 90
```

Raises type error on nonnumeric *m*.

Raises type error on nonnumeric *n*.

Returns nonnegative integer.

### 15.2.17 `mathtools.greatest_power_of_two_less_equal`

`mathtools.greatest_power_of_two_less_equal(n, i=0)`

Greatest integer power of two less than or equal to positive *n*.

```
>>> for n in range(10, 20):
...     print '\t%s\t%s' % (n, mathtools.greatest_power_of_two_less_equal(n))
...
10 8
11 8
12 8
13 8
14 8
15 8
16 16
17 16
18 16
19 16
```

Greatest-but-*i* integer power of 2 less than or equal to positive *n*:

```
>>> for n in range(10, 20):
...     print '\t%s\t%s' % (n, mathtools.greatest_power_of_two_less_equal(n, i=1))
...
10 4
11 4
12 4
13 4
14 4
15 4
16 8
17 8
18 8
19 8
```

Raises type error on nonnumeric *n*.

Raises value error on nonpositive *n*.

Returns positive integer.

### 15.2.18 `mathtools.integer_equivalent_number_to_integer`

`mathtools.integer_equivalent_number_to_integer(number)`

Integer-equivalent *number* to integer.

```
>>> mathtools.integer_equivalent_number_to_integer(17.0)
17
```

Returns noninteger-equivalent number unchanged:

```
>>> mathtools.integer_equivalent_number_to_integer(17.5)
17.5
```

Raises type error on nonnumber input.

Returns number.

### 15.2.19 `mathtools.integer_to_base_k_tuple`

`mathtools.integer_to_base_k_tuple(n, k)`  
Nonnegative integer  $n$  to base- $k$  tuple.

```
>>> mathtools.integer_to_base_k_tuple(1066, 10)
(1, 0, 6, 6)
```

Returns tuple of one or more positive integers.

### 15.2.20 `mathtools.integer_to_binary_string`

`mathtools.integer_to_binary_string(n)`  
Positive integer  $n$  to binary string.

```
>>> for n in range(1, 16 + 1):
...     print '{}\t{}'.format(n, mathtools.integer_to_binary_string(n))
...
1  1
2  10
3  11
4  100
5  101
6  110
7  111
8  1000
9  1001
10 1010
11 1011
12 1100
13 1101
14 1110
15 1111
16 10000
```

Returns string.

### 15.2.21 `mathtools.interpolate_cosine`

`mathtools.interpolate_cosine(y1, y2, mu)`  
Cosine interpolate  $y1$  and  $y2$  with  $mu$  normalized  $[0, 1]$ .

```
>>> mathtools.interpolate_cosine(0, 1, 0.5)
0.49999999999999994
```

Returns float.

### 15.2.22 `mathtools.interpolate_divide`

`mathtools.interpolate_divide(total, start_fraction, stop_fraction, exp='cosine')`  
Divide *total* into segments of sizes computed from interpolating between *start\_fraction* and *stop\_fraction*:

```
>>> mathtools.interpolate_divide(10, 1, 1, exp=1)
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> sum(mathtools.interpolate_divide(10, 1, 1, exp=1))
10.0
```

```
>>> mathtools.interpolate_divide(10, 5, 1)
[4.7986734489043181, 2.8792040693425909, 1.3263207210948171,
0.99580176065827419]
>>> sum(mathtools.interpolate_divide(10, 5, 1))
10.0
```

Set `exp='cosine'` for cosine interpolation.

Set `exp` to a numeric value for exponential interpolation with `exp` as the exponent.

Scale resulting segments so that their sum equals exactly *total*.

Returns a list of floats.

### 15.2.23 `mathtools.interpolate_divide_multiple`

`mathtools.interpolate_divide_multiple(totals, key_values, exp='cosine')`

Interpolate *key\_values* such that the sum of the resulting interpolated values equals the given *totals*:

```
>>> mathtools.interpolate_divide_multiple([100, 50], [20, 10, 20])
[19.4487, 18.5201, 16.2270, 13.7156, 11.7488, 10.4879,
9.8515, 9.5130, 10.4213, 13.0736, 16.9918]
```

The operation is the same as `mathtools.interpolate_divide()`. But this function takes multiple *totals* and *key\_values* at once.

Precondition: `len(totals) == len(key_values) - 1`.

Set *totals* equal to a list or tuple of the total sum of interpolated values.

Set *key\_values* equal to a list or tuple of key values to interpolate.

Set `exp` to *consine* for consine interpolation.

Set `exp` to a number for exponential interpolation.

Returns a list of floats.

### 15.2.24 `mathtools.interpolate_exponential`

`mathtools.interpolate_exponential(y1, y2, mu, exp=1)`

Exponential interpolate *y1* and *y2* with *mu* normalized `[0, 1]`.

```
>>> mathtools.interpolate_exponential(0, 1, 0.5, 4)
0.0625
```

Set `exp` equal to the exponent of interpolation.

Returns float.

### 15.2.25 `mathtools.interpolate_linear`

`mathtools.interpolate_linear(y1, y2, mu)`

Linear interpolate *y1* and *y2* with *mu* normalized `[0, 1]`.

```
>>> mathtools.interpolate_linear(0, 1, 0.5)
0.5
```

Returns float.

### 15.2.26 `mathtools.interval_string_to_pair_and_indicators`

`mathtools.interval_string_to_pair_and_indicators` (*interval\_string*)  
 Changes *interval\_string* to pair, boolean start indicator and boolean stop indicator.

```
>>> mathtools.interval_string_to_pair_and_indicators('[5, 8)')
((5, 8), False, True)
```

Parses square brackets as closed interval bounds.

Parses parentheses as open interval bounds.

Returns triple.

### 15.2.27 `mathtools.is_assignable_integer`

`mathtools.is_assignable_integer` (*expr*)  
 True when *expr* is equivalent to an integer and can be written without recourse to ties.

```
>>> for n in range(0, 16 + 1):
...     print '%s\t%s' % (n, mathtools.is_assignable_integer(n))
...
0 False
1 True
2 True
3 True
4 True
5 False
6 True
7 True
8 True
9 False
10 False
11 False
12 True
13 False
14 True
15 True
16 True
```

Otherwise false.

Returns boolean.

### 15.2.28 `mathtools.is_dotted_integer`

`mathtools.is_dotted_integer` (*expr*)  
 True when *expr* is equivalent to a positive integer and can be written with zero or more dots.

```
>>> for expr in range(16):
...     print '%s\t%s' % (expr, mathtools.is_dotted_integer(expr))
...
0 False
1 False
2 False
3 True
4 False
5 False
6 True
7 True
8 False
9 False
10 False
11 False
12 True
13 False
14 True
15 True
```

Otherwise false.

Returns boolean.

Integer  $n$  qualifies as dotted when  $\text{abs}(n)$  is of the form  $2^{**j} * (2^{**k} - 1)$  with integers  $0 \leq j$ ,  $2 < k$ .

### 15.2.29 `mathtools.is_integer_equivalent_expr`

`mathtools.is_integer_equivalent_expr(expr)`

True when *expr* is an integer-equivalent number.

```
>>> mathtools.is_integer_equivalent_expr(12.0)
True
```

True when *expr* evaluates to an integer:

```
>>> mathtools.is_integer_equivalent_expr('12')
True
```

Otherwise false:

```
>>> mathtools.is_integer_equivalent_expr('foo')
False
```

Returns boolean.

### 15.2.30 `mathtools.is_integer_equivalent_number`

`mathtools.is_integer_equivalent_number(expr)`

True when *expr* is a number and *expr* is equivalent to an integer.

```
>>> mathtools.is_integer_equivalent_number(12.0)
True
```

Otherwise false:

```
>>> mathtools.is_integer_equivalent_number(Duration(1, 2))
False
```

Returns boolean.

### 15.2.31 `mathtools.is_negative_integer`

`mathtools.is_negative_integer(expr)`

True when *expr* equals a negative integer.

```
>>> mathtools.is_negative_integer(-1)
True
```

Otherwise false:

```
>>> mathtools.is_negative_integer(0)
False
```

```
>>> mathtools.is_negative_integer(99)
False
```

Returns boolean.

### 15.2.32 `mathtools.is_nonnegative_integer`

`mathtools.is_nonnegative_integer` (*expr*)  
True when *expr* equals a nonnegative integer.

```
>>> mathtools.is_nonnegative_integer(99)
True
```

```
>>> mathtools.is_nonnegative_integer(0)
True
```

Otherwise false:

```
>>> mathtools.is_nonnegative_integer(-1)
False
```

Returns boolean.

### 15.2.33 `mathtools.is_nonnegative_integer_equivalent_number`

`mathtools.is_nonnegative_integer_equivalent_number` (*expr*)  
True when *expr* is a nonnegative integer-equivalent number. Otherwise false:

```
>>> mathtools.is_nonnegative_integer_equivalent_number(Duration(4, 2))
True
```

Returns boolean.

### 15.2.34 `mathtools.is_nonnegative_integer_power_of_two`

`mathtools.is_nonnegative_integer_power_of_two` (*expr*)  
True when *expr* is a nonnegative integer power of 2.

```
>>> for n in range(10):
...     print n, mathtools.is_nonnegative_integer_power_of_two(n)
...
0 True
1 True
2 True
3 False
4 True
5 False
6 False
7 False
8 True
9 False
```

Otherwise false.

Returns boolean.

### 15.2.35 `mathtools.is_positive_integer`

`mathtools.is_positive_integer` (*expr*)  
True when *expr* equals a positive integer.

```
>>> mathtools.is_positive_integer(99)
True
```

Otherwise false:

```
>>> mathtools.is_positive_integer(0)
False
```

```
>>> mathtools.is_positive_integer(-1)
False
```

Returns boolean.

### 15.2.36 `mathtools.is_positive_integer_equivalent_number`

`mathtools.is_positive_integer_equivalent_number(expr)`  
True when *expr* is a positive integer-equivalent number. Otherwise false:

```
>>> mathtools.is_positive_integer_equivalent_number(Duration(4, 2))
True
```

Returns boolean.

### 15.2.37 `mathtools.is_positive_integer_power_of_two`

`mathtools.is_positive_integer_power_of_two(expr)`  
True when *expr* is a positive integer power of 2.

```
>>> for n in range(10):
...     print n, mathtools.is_positive_integer_power_of_two(n)
...
0 False
1 True
2 True
3 False
4 True
5 False
6 False
7 False
8 True
9 False
```

Otherwise false.

Returns boolean.

### 15.2.38 `mathtools.least_common_multiple`

`mathtools.least_common_multiple(*integers)`  
Least common multiple of positive *integers*.

```
>>> mathtools.least_common_multiple(2, 4, 5, 10, 20)
20
```

Returns positive integer.

### 15.2.39 `mathtools.least_multiple_greater_equal`

`mathtools.least_multiple_greater_equal(m, n)`  
Returns the least integer multiple of *m* greater than or equal to *n*.

```
>>> mathtools.least_multiple_greater_equal(10, 47)
50
```

```
>>> for m in range(1, 10):
...     print m, mathtools.least_multiple_greater_equal(m, 47)
...
1 47
2 48
3 48
```



```

4 48
5 50
6 48
7 49
8 48
9 54

```

```

>>> for n in range(10, 100, 10):
...     print mathtools.least_multiple_greater_equal(7, n), n
...
14 10
21 20
35 30
42 40
56 50
63 60
70 70
84 80
91 90

```

Returns integer.

### 15.2.40 `mathtools.least_power_of_two_greater_equal`

`mathtools.least_power_of_two_greater_equal(n, i=0)`

Returns least integer power of two greater than or equal to positive  $n$ .

```

>>> for n in range(10, 20):
...     print '\t%s\t%s' % (n, mathtools.least_power_of_two_greater_equal(n))
...
10 16
11 16
12 16
13 16
14 16
15 16
16 16
17 32
18 32
19 32

```

When  $i = 1$ , returns the first integer power of 2 greater than the least integer power of 2 greater than or equal to  $n$ .

```

>>> for n in range(10, 20):
...     print '\t%s\t%s' % (n, mathtools.least_power_of_two_greater_equal(n, i=1))
...
10 32
11 32
12 32
13 32
14 32
15 32
16 32
17 64
18 64
19 64

```

When  $i = 2$ , returns the second integer power of 2 greater than the least integer power of 2 greater than or equal to  $n$ , and, in general, return the  $i$ th integer power of 2 greater than the least integer power of 2 greater than or equal to  $n$ .

Raises type error on nonnumeric  $n$ .

Raises value error on nonpositive  $n$ .

Returns integer.

### 15.2.41 `mathtools.next_integer_partition`

`mathtools.next_integer_partition(integer_partition)`

Next integer partition following *integer\_partition* in descending lex order.

```
>>> mathtools.next_integer_partition((8, 3))
(8, 2, 1)
```

```
>>> mathtools.next_integer_partition((8, 2, 1))
(8, 1, 1, 1)
```

```
>>> mathtools.next_integer_partition((8, 1, 1, 1))
(7, 4)
```

Input *integer\_partition* must be sequence of positive integers.

Returns integer partition as tuple of positive integers.

### 15.2.42 `mathtools.partition_integer_by_ratio`

`mathtools.partition_integer_by_ratio(n, ratio)`

Partitions positive integer-equivalent *n* by *ratio*.

```
>>> mathtools.partition_integer_by_ratio(10, [1, 2])
[3, 7]
```

Partitions positive integer-equivalent *n* by *ratio* with negative parts:

```
>>> mathtools.partition_integer_by_ratio(10, [1, -2])
[3, -7]
```

Partitions negative integer-equivalent *n* by *ratio*:

```
>>> mathtools.partition_integer_by_ratio(-10, [1, 2])
[-3, -7]
```

Partitions negative integer-equivalent *n* by *ratio* with negative parts:

```
>>> mathtools.partition_integer_by_ratio(-10, [1, -2])
[-3, 7]
```

Returns result with weight equal to absolute value of *n*.

Raises type error on noninteger *n*.

Returns list of integers.

### 15.2.43 `mathtools.partition_integer_into_canonic_parts`

`mathtools.partition_integer_into_canonic_parts(n, decrease_parts_monotonically=True)`

Partitions integer *n* into canonic parts.

Returns all parts positive on positive *n*:

```
>>> for n in range(1, 11):
...     print n, mathtools.partition_integer_into_canonic_parts(n)
...
1 (1,)
2 (2,)
3 (3,)
4 (4,)
5 (4, 1)
6 (6,)
7 (7,)
8 (8,)
```

```
9 (8, 1)
10 (8, 2)
```

Returns all parts negative on negative  $n$ :

```
>>> for n in reversed(range(-20, -10)):
...     print n, mathtools.partition_integer_into_canonic_parts(n)
...
-11 (-8, -3)
-12 (-12,)
-13 (-12, -1)
-14 (-14,)
-15 (-15,)
-16 (-16,)
-17 (-16, -1)
-18 (-16, -2)
-19 (-16, -3)
-20 (-16, -4)
```

Returns parts that increase monotonically:

```
>>> for n in range(11, 21):
...     print n, mathtools.partition_integer_into_canonic_parts(n,
...         decrease_parts_monotonically=False)
...
11 (3, 8)
12 (12,)
13 (1, 12)
14 (14,)
15 (15,)
16 (16,)
17 (1, 16)
18 (2, 16)
19 (3, 16)
20 (4, 16)
```

Returns tuple with parts that decrease monotonically.

Raises type error on noninteger  $n$ .

Returns tuple of one or more integers.

### 15.2.44 mathtools.partition\_integer\_into\_halves

`mathtools.partition_integer_into_halves( $n$ ,  $bigger='left'$ ,  $even='allowed'$ )`

Writes positive integer  $n$  as the pair  $t = (left, right)$  such that  $n == left + right$ .

When  $n$  is odd the greater part of  $t$  corresponds to the value of *bigger*:

```
>>> mathtools.partition_integer_into_halves(7, bigger='left')
(4, 3)
>>> mathtools.partition_integer_into_halves(7, bigger='right')
(3, 4)
```

Likewise when  $n$  is even and  $even = 'disallowed'$ :

```
>>> mathtools.partition_integer_into_halves(8, bigger='left', even='disallowed')
(5, 3)
>>> mathtools.partition_integer_into_halves(8, bigger='right', even='disallowed')
(3, 5)
```

But when  $n$  is even and  $even = 'allowed'$  then  $left == right$  and *bigger* is ignored:

```
>>> mathtools.partition_integer_into_halves(8)
(4, 4)
>>> mathtools.partition_integer_into_halves(8, bigger='left')
(4, 4)
>>> mathtools.partition_integer_into_halves(8, bigger='right')
(4, 4)
```

When  $n$  is 0 return (0, 0):

```
>>> mathtools.partition_integer_into_halves(0)
(0, 0)
```

When  $n$  is 0 and `even = 'disallowed'` raises partition error.

Raises type error on noninteger  $n$ .

Raises value error on negative  $n$ .

Returns pair of positive integers.

### 15.2.45 `mathtools.partition_integer_into_parts_less_than_double`

`mathtools.partition_integer_into_parts_less_than_double(n, m)`

Partitions integer  $n$  into parts less than double integer  $m$ .

**Example:**

```
>>> for n in range(1, 24+1):
...     print n, mathtools.partition_integer_into_parts_less_than_double(n, 4)
1 (1,)
2 (2,)
3 (3,)
4 (4,)
5 (5,)
6 (6,)
7 (7,)
8 (4, 4)
9 (4, 5)
10 (4, 6)
11 (4, 7)
12 (4, 4, 4)
13 (4, 4, 5)
14 (4, 4, 6)
15 (4, 4, 7)
16 (4, 4, 4, 4)
17 (4, 4, 4, 5)
18 (4, 4, 4, 6)
19 (4, 4, 4, 7)
20 (4, 4, 4, 4, 4)
21 (4, 4, 4, 4, 5)
22 (4, 4, 4, 4, 6)
23 (4, 4, 4, 4, 7)
24 (4, 4, 4, 4, 4, 4)
```

Returns tuple of one or more integers.

### 15.2.46 `mathtools.partition_integer_into_units`

`mathtools.partition_integer_into_units(n)`

Partitions positive integer into units:

```
>>> mathtools.partition_integer_into_units(6)
[1, 1, 1, 1, 1, 1]
```

Partitions negative integer into units:

```
>>> mathtools.partition_integer_into_units(-5)
[-1, -1, -1, -1, -1]
```

Partitions 0 into units:

```
>>> mathtools.partition_integer_into_units(0)
[]
```

Returns list of zero or more parts with absolute value equal to 1.

### 15.2.47 `mathtools.remove_powers_of_two`

`mathtools.remove_powers_of_two(n)`

Removes powers of 2 from the factors of positive integer  $n$ :

```
>>> for n in range(10, 100, 10):
...     print '\t%s\t%s' % (n, mathtools.remove_powers_of_two(n))
...
    10 5
    20 5
    30 15
    40 5
    50 25
    60 15
    70 35
    80 5
    90 45
```

Raises type error on noninteger  $n$ .

Raises value error on nonpositive  $n$ .

Returns positive integer.

### 15.2.48 `mathtools.sign`

`mathtools.sign(n)`

Returns  $-1$  on negative  $n$ :

```
>>> mathtools.sign(-96.2)
-1
```

Returns 0 when  $n$  is 0:

```
>>> mathtools.sign(0)
0
```

Returns 1 on positive  $n$ :

```
>>> mathtools.sign(Duration(9, 8))
1
```

Returns  $-1$ , 0 or 1.

### 15.2.49 `mathtools.weight`

`mathtools.weight(sequence)`

Sum of the absolute value of the elements in *sequence*:

```
>>> mathtools.weight([-1, -2, 3, 4, 5])
15
```

Returns nonnegative integer.

### 15.2.50 `mathtools.yield_all_compositions_of_integer`

`mathtools.yield_all_compositions_of_integer(n)`

Yields all compositions of positive integer  $n$  in descending lex order:

```
>>> for integer_composition in mathtools.yield_all_compositions_of_integer(5):
...     integer_composition
...
(5,)
(4, 1)
(3, 2)
```

```
(3, 1, 1)
(2, 3)
(2, 2, 1)
(2, 1, 2)
(2, 1, 1, 1)
(1, 4)
(1, 3, 1)
(1, 2, 2)
(1, 2, 1, 1)
(1, 1, 3)
(1, 1, 2, 1)
(1, 1, 1, 2)
(1, 1, 1, 1, 1)
```

Integer compositions are ordered integer partitions.

Returns generator of positive integer tuples of length at least 1.

### 15.2.51 `mathtools.yield_all_partitions_of_integer`

`mathtools.yield_all_partitions_of_integer(n)`

Yields all partitions of positive integer  $n$  in descending lex order:

```
>>> for partition in mathtools.yield_all_partitions_of_integer(7):
...     partition
...
(7,)
(6, 1)
(5, 2)
(5, 1, 1)
(4, 3)
(4, 2, 1)
(4, 1, 1, 1)
(3, 3, 1)
(3, 2, 2)
(3, 2, 1, 1)
(3, 1, 1, 1, 1)
(2, 2, 2, 1)
(2, 2, 1, 1, 1)
(2, 1, 1, 1, 1, 1)
(1, 1, 1, 1, 1, 1, 1)
```

Returns generator of positive integer tuples of length at least 1.

### 15.2.52 `mathtools.yield_nonreduced_fractions`

`mathtools.yield_nonreduced_fractions()`

Yields positive nonreduced fractions in Cantor diagonalized order:

```
>>> generator = mathtools.yield_nonreduced_fractions()
>>> for n in range(16):
...     generator.next()
...
(1, 1)
(2, 1)
(1, 2)
(1, 3)
(2, 2)
(3, 1)
(4, 1)
(3, 2)
(2, 3)
(1, 4)
(1, 5)
(2, 4)
(3, 3)
(4, 2)
```

```
(5, 1)
(6, 1)
```

Returns generator.

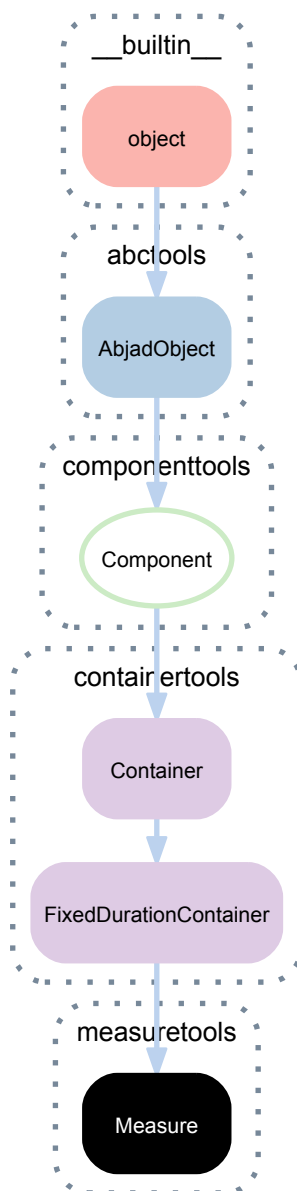




# MEASURETOOLS

## 16.1 Concrete classes

### 16.1.1 measuretools.Measure



**class** `measuretools.Measure` (*time\_signature*, *music=None*, *\*\*kwargs*)  
A measure.

```
>>> measure = Measure((4, 8), "c'8 d' e' f'")
>>> show(measure)
```



## Bases

- `containertools.FixedDurationContainer`
- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

**Measure.has\_non\_power\_of\_two\_denominator**

True when measure time signature denominator is not an integer power of 2:

```
>>> measure = Measure((5, 9), "c'8 d' e' f' g'")
>>> measure.has_non_power_of_two_denominator
True
```

Otherwise false:

```
>>> measure = Measure((5, 8), "c'8 d' e' f' g'")
>>> measure.has_non_power_of_two_denominator
False
```

Returns boolean.

**Measure.has\_power\_of\_two\_denominator**

True when measure time signature denominator is an integer power of 2:

```
>>> measure = Measure((5, 8), "c'8 d' e' f' g'")
>>> measure.has_power_of_two_denominator
True
```

Otherwise false:

```
>>> measure = Measure((5, 9), "c'8 d' e' f' g'")
>>> measure.has_power_of_two_denominator
False
```

Returns boolean.

**Measure.implied\_prolation**

Implied prololation of measure time signature:

```
>>> measure = Measure((5, 12), "c'8 d' e' f' g'")
```

```
>>> measure.implied_prolation
Multiplier(2, 3)
```

Returns multiplier.

**Measure.is\_full**

True when prolated duration equals time signature duration:

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
```

```
>>> measure.is_full
True
```

Otherwise false.

Returns boolean.

#### **Measure.is\_misfilled**

True when measure is either underfull or overfull:

```
>>> measure = Measure((3, 4), "c' d' e' f'")
```

```
>>> measure
Measure(3/4, [c'4, d'4, e'4, f'4])
```

```
>>> measure.is_misfilled
True
```

Otherwise false:

```
>>> measure = Measure((3, 4), "c' d' e'")
```

```
>>> measure
Measure(3/4, [c'4, d'4, e'4])
```

```
>>> measure.is_misfilled
False
```

Returns boolean.

#### **Measure.is\_overfull**

True when prolated duration is greater than time signature duration:

```
>>> measure = Measure((3, 4), "c'4 d' e' f'")
```

```
>>> measure.is_overfull
True
```

Otherwise false.

Returns boolean.

#### **Measure.is\_underfull**

True when prolated duration is less than time signature duration:

```
>>> measure = Measure((3, 4), "c'4 d'")
```

```
>>> measure.is_underfull
True
```

Otherwise false.

Returns boolean.

#### **Measure.lilypond\_format**

#### **Measure.measure\_number**

1-indexed measure number:

```
>>> staff = Staff()
>>> staff.append(Measure((3, 4), "c' d' e'"))
>>> staff.append(Measure((2, 4), "f' g'"))
```

```
>>> staff[0].measure_number
1
```

```
>>> staff[1].measure_number
2
```

Returns positive integer.

(Component) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Component) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

(Component) **.storage\_format**

Storage format of component.

Returns string.

Measure **.target\_duration**

Target duration of measure always equal to duration of effective time signature.

Returns duration.

Measure **.time\_signature**

Effective time signature of measure.

Returns time signature or none.

## Read/write properties

Measure **.always\_format\_time\_signature**

Read / write flag to indicate whether time signature should appear in LilyPond format even when not expected.

Set to true when necessary to print the same signature repeatedly.

Default to false.

Returns boolean.

Measure **.automatically\_adjust\_time\_signature**

Read / write flag to indicate whether time signature should update automatically following contents-changing operations:

```
>>> measure = Measure((3, 4), "c' d' e'")
```

```
>>> measure
Measure(3/4, [c'4, d'4, e'4])
```

```
>>> measure.automatically_adjust_time_signature = True
>>> measure.append('r')
```

```
>>> measure
Measure(4/4, [c'4, d'4, e'4, r4])
```

Default to false.

Returns boolean.

(Container) **.is\_simultaneous**

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

## Methods

(Container) **.append**(*component*)  
 Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) **.extend**(*expr*)  
 Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```

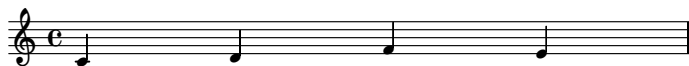


Returns none.

(Container) **.index** (*component*)

Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e' 4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

```
(Container).insert(i, component, fracture_spanners=False)
```

Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs'16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```

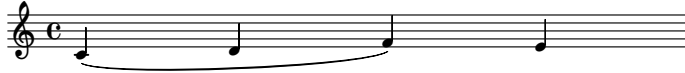


Returns none.

(Container) **.pop** (*i=-1*)

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

(Container) **.remove** (*component*)

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse** ()

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

Measure.**scale\_and\_adjust\_time\_signature** (*multiplier=None*)

Scales *measure* by *multiplier* and adjusts time signature:

**Example 1.** Scale measure by non-power-of-two multiplier:

```
>>> measure = Measure((3, 8), "c'8 d'8 e'8")
>>> show(measure)
```



```
>>> measure.scale_and_adjust_time_signature(Multiplier(2, 3))
>>> show(measure)
```



Returns none.

(Component) **.select** (*sequential=False*)

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)

Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords** ()

Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

Measure **.\_\_add\_\_** (*arg*)

Add two measures together in-score or outside-of-score.

Wrapper around `measuretools.fuse_measures()`.

(Container) **.\_\_contains\_\_** (*expr*)

True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component) **.\_\_copy\_\_** (*\*args*)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

Measure **.\_\_delitem\_\_** (*i*)

Container item deletion with optional time signature adjustment.

(AbjadObject) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container) **.\_\_getitem\_\_** (*i*)

Get container *i*. Shallow traversal of container for numeric indices only.



Returns component.

(Container) .**\_\_len\_\_**()  
Number of items in container.

Returns nonnegative integer.

(Component) .**\_\_mul\_\_**(*n*)  
Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

Measure .**\_\_repr\_\_**()  
Interpreter representation of measure.

Returns string.

(Component) .**\_\_rmul\_\_**(*n*)  
Copies component *n* times and detach spanners.

Returns list of new components.

Measure .**\_\_setitem\_\_**(*i*, *expr*)  
Container setitem logic with optional time signature adjustment.

Measure setitem logic now adjusts time signature automatically when `adjust_time_signature_automatically` is true.

Measure .**\_\_str\_\_**()  
String form of measure with pipes for single string display.

## 16.2 Functions

### 16.2.1 `measuretools.append_spacer_skip_to_underfull_measure`

`measuretools.append_spacer_skip_to_underfull_measure`(*measure*)  
Append spacer skip to underfull *measure*:

```
>>> measure = Measure((4, 12), "c'8 d'8 e'8 f'8")
>>> time_signature = inspect(measure).get_mark(
...     contexttools.TimeSignatureMark)
>>> time_signature.detach()
TimeSignatureMark((4, 12))
>>> new_time_signature = contexttools.TimeSignatureMark((5, 12))
>>> new_time_signature.attach(measure)
TimeSignatureMark((5, 12))(|5/12 c'8 d'8 e'8 f'8|)
>>> measure.is_underfull
True
```

```
>>> measuretools.append_spacer_skip_to_underfull_measure(measure)
Measure(5/12, [c'8, d'8, e'8, f'8, s1 * 1/8])
```

Append nothing to nonunderfull *measure*.

Return *measure*.

### 16.2.2 `measuretools.append_spacer_skips_to_underfull_measures_in_expr`

`measuretools.append_spacer_skips_to_underfull_measures_in_expr`(*expr*)  
Append spacer skips to underfull measures in *expr*:

```
>>> staff = Staff(Measure((3, 8), "c'8 d'8 e'8") * 3)
>>> time_signature = inspect(staff[1]).get_mark(
...     contexttools.TimeSignatureMark)
>>> time_signature.detach()
TimeSignatureMark((3, 8))
>>> new_time_signature = contexttools.TimeSignatureMark((4, 8))
>>> new_time_signature.attach(staff[1])
TimeSignatureMark((4, 8))(|4/8 c'8 d'8 e'8|)
>>> time_signature = inspect(
...     staff[2]).get_mark(contexttools.TimeSignatureMark)
>>> time_signature.detach()
TimeSignatureMark((3, 8))
>>> new_time_signature = contexttools.TimeSignatureMark((5, 8))
>>> new_time_signature.attach(staff[2])
TimeSignatureMark((5, 8))(|5/8 c'8 d'8 e'8|)
>>> staff[1].is_underfull
True
>>> staff[2].is_underfull
True
```

```
>>> measuretools.append_spacer_skips_to_underfull_measures_in_expr(staff)
[Measure(4/8, [c'8, d'8, e'8, s1 * 1/8]), Measure(5/8, [c'8, d'8, e'8, s1 * 1/4])]
```

Returns measures treated.

### 16.2.3 measuretools.apply\_full\_measure\_tuplets\_to\_contents\_of\_measures\_in\_expr

`measuretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr` (*expr*,  
*sup-ple-ment=None*)

Applies full-measure tuplets to contents of measures in *expr*:

```
>>> staff = Staff([
...     Measure((2, 8), "c'8 d'8"),
...     Measure((3, 8), "e'8 f'8 g'8")])
>>> show(staff)
```



```
>>> measuretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr(staff)
>>> show(staff)
```



Returns none.

### 16.2.4 measuretools.extend\_measures\_in\_expr\_and\_apply\_full\_measure\_tuplets

`measuretools.extend_measures_in_expr_and_apply_full_measure_tuplets` (*expr*,  
*sup-ple-ment*)

Extend measures in *expr* with *supplement* and apply full-measure tuplets to contents of measures:

```
>>> staff = Staff([Measure((2, 8), "c'8 d'8"), Measure((3, 8), "e'8 f'8 g'8")])
```

```
>>> supplement = [Rest((1, 16))]
>>> measuretools.extend_measures_in_expr_and_apply_full_measure_tuplets(
...     staff, supplement)
```

Returns none.

### 16.2.5 `measuretools.fill_measures_in_expr_with_full_measure_spacer_skips`

[illegible]

Fill measures in *expr* with full-measure spacer skips.

### 16.2.6 `measuretools.fill_measures_in_expr_with_minimal_number_of_notes`

[illegible]

Fill measures in *expr* with minimal number of notes that decrease durations monotonically:

```
>>> measure = Measure((5, 18), [])
```

```
>>> measuretools.fill_measures_in_expr_with_minimal_number_of_notes(
...     measure, decrease_durations_monotonically=True)
```

Fill measures in *expr* with minimal number of notes that increase durations monotonically:

```
>>> measure = Measure((5, 18), [])
```

```
>>> measuretools.fill_measures_in_expr_with_minimal_number_of_notes(
...     measure, decrease_durations_monotonically=False)
```

Returns none.

### 16.2.7 `measuretools.fill_measures_in_expr_with_repeated_notes`

[illegible]

Fill measures in *expr* with repeated notes.

### 16.2.8 `measuretools.fill_measures_in_expr_with_time_signature_denominator_notes`

[illegible]

Fill measures in *expr* with time signature denominator notes:

```
>>> staff = Staff([Measure((3, 4), []), Measure((3, 16), []), Measure((3, 8), [])])
>>> measuretools.fill_measures_in_expr_with_time_signature_denominator_notes(staff)
```

Delete existing contents of measures in *expr*.

Returns none.

### 16.2.9 `measuretools.get` measure that starts with container

`measuretools.get_measure_that_starts_with_container` (*container*)  
Get measure that starts with *container*.

Returns measure or none.

### 16.2.10 `measuretools.get_measure_that_stops_with_container`

`measuretools.get_measure_that_stops_with_container` (*container*)

Get measure that stops with *container*.

Returns measure or none.

### 16.2.11 `measuretools.get_next_measure_from_component`

`measuretools.get_next_measure_from_component` (*component*)

Get next measure from *component*.

When *component* is a voice, staff or other sequential context, and when *component* contains a measure, return first measure in *component*. This starts the process of forwards measure iteration.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> measuretools.get_next_measure_from_component(staff)
Measure(2/8, [c'8, d'8])
```

When *component* is voice, staff or other sequential context, and when *component* contains no measure, raise missing measure error.

When *component* is a measure and there is a measure immediately following *component*, return measure immediately following component.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> measuretools.get_previous_measure_from_component(staff[0]) is None
True
```

When *component* is a measure and there is no measure immediately following *component*, return None.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> measuretools.get_previous_measure_from_component(staff[-1])
Measure(2/8, [c'8, d'8])
```

When *component* is a leaf and there is a measure in the parentage of *component*, return the measure in the parentage of *component*.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> measuretools.get_previous_measure_from_component(staff.select_leaves()[0])
Measure(2/8, [c'8, d'8])
```

When *component* is a leaf and there is no measure in the parentage of *component*, raise missing measure error.

### 16.2.12 `measuretools.get_one_indexed_measure_number_in_expr`

`measuretools.get_one_indexed_measure_number_in_expr` (*expr*, *measure\_number*)

Gets one-indexed *measure\_number* in *expr*.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 || 2/8 g'8 a'8 |")
>>> show(staff)
```



```
>>> measuretools.get_one_indexed_measure_number_in_expr(staff, 3)
Measure(2/8, [g'8, a'8])
```

Note that measures number from 1.

### 16.2.13 `measuretools.get_previous_measure_from_component`

`measuretools.get_previous_measure_from_component` (*component*)

Get previous measure from *component*.

When *component* is voice, staff or other sequential context, and when *component* contains a measure, return last measure in *component*. This starts the process of backwards measure iteration.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> measuretools.get_previous_measure_from_component(staff)
Measure(2/8, [e'8, f'8])
```

When *component* is voice, staff or other sequential context, and when *component* contains no measure, raise missing measure error.

When *component* is a measure and there is a measure immediately preceeding *component*, return measure immediately preceeding *component*.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> measuretools.get_previous_measure_from_component(staff[-1])
Measure(2/8, [c'8, d'8])
```

When *component* is a measure and there is no measure immediately preceeding *component*, return None.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> measuretools.get_previous_measure_from_component(staff[0]) is None
True
```

When *component* is a leaf and there is a measure in the parentage of *component*, return the measure in the parentage of *component*.

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> measuretools.get_previous_measure_from_component(staff.select_leaves()[0])
Measure(2/8, [c'8, d'8])
```

When *component* is a leaf and there is no measure in the parentage of *component*, raise missing measure error.

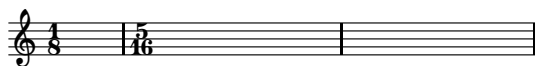
### 16.2.14 `measuretools.make_measures_with_full_measure_spacer_skips`

`measuretools.make_measures_with_full_measure_spacer_skips` (*time\_signatures*)

Make measures with full-measure spacer skips from *time\_signatures*.

**Example.**

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> staff = Staff(measures)
>>> show(staff)
```



Returns selection of unincorporated measures.

### 16.2.15 `measuretools.move_full_measure_tuplet_prolation_to_measure_time_signature`

`measuretools.move_full_measure_tuplet_prolation_to_measure_time_signature` (*expr*)

Move prolotion of full-measure tuplet to time signature of measure.

Measures usually become non-power-of-two as as result:

```
>>> t = Measure((2, 8), [tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")])
>>> measuretools.move_full_measure_tuplet_prolation_to_measure_time_signature(t)
```

Returns none.

### 16.2.16 `measuretools.move_measure_prolation_to_full_measure_tuplet`

`measuretools.move_measure_prolation_to_full_measure_tuplet` (*expr*)

Move measure prolotion to full-measure tuplet.

Turn non-power-of-two measures into power-of-two measures containing a single fixed-duration tuplet.

Note that not all non-power-of-two measures can be made power-of-two.

Returns None because processes potentially many measures.

### 16.2.17 `measuretools.replace_contents_of_measures_in_expr`

`measuretools.replace_contents_of_measures_in_expr` (*expr*, *new\_contents*)

Replaces contents of measures in *expr* with *new\_contents*.

**Example.**

```
>>> staff = Staff(measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (3, 16)]))
>>> show(staff)
```



```
>>> notes = [Note("c'16"), Note("d'16"), Note("e'16"), Note("f'16")]
>>> measuretools.replace_contents_of_measures_in_expr(staff, notes)
[Measure(1/8, [c'16, d'16]), Measure(3/16, [e'16, f'16, s1 * 1/16])]
>>> show(staff)
```



Preserves duration of all measures.

Skips measures that are too small.

Pads extra space at end of measures with spacer skip.

Raises stop iteration if not enough measures.

Returns measures iterated.

### 16.2.18 `measuretools.scale_measure_denominator_and_adjust_measure_contents`

`measuretools.scale_measure_denominator_and_adjust_measure_contents` (*measure*, *factor*)

Scales power-of-two *measure* to non-power-of-two measure with new denominator *factor*:

**Example.**

```
>>> measure = Measure((2, 8), "c'8 d'8")
>>> beam = spannertools.BeamSpanner()
>>> beam.attach(measure.select_leaves())
>>> show(measure)
```



```
>>> measuretools.scale_measure_denominator_and_adjust_measure_contents(
...     measure, 3)
Measure(3/12, [c'8., d'8.])
>>> show(measure)
```



Treats new denominator *factor* like clever form of 1: 3/3 or 5/5 or 7/7, etc.

Preserves *measure* prolated duration.

Derives new *measure* multiplier.

Scales *measure* contents.

Picks best new time signature.

### 16.2.19 `measuretools.set_always_format_time_signature_of_measures_in_expr`

`measuretools.set_always_format_time_signature_of_measures_in_expr` (*expr*,  
*value=True*)

Set *always\_format\_time\_signature* of measures in *expr* to boolean *value*.

Returns none.

### 16.2.20 `measuretools.set_measure_denominator_and_adjust_numerator`

`measuretools.set_measure_denominator_and_adjust_numerator` (*measure*, *denomi-*  
*nator*)

Set *measure* time signature *denominator* and multiply time signature numerator accordingly:

```
>>> measure = Measure((3, 8), "c'8 d'8 e'8")
>>> spannertools.BeamSpanner(measure.select_leaves())
BeamSpanner(c'8, d'8, e'8)
```

```
>>> measuretools.set_measure_denominator_and_adjust_numerator(measure, 16)
Measure(6/16, [c'8, d'8, e'8])
```

Leave *measure* contents unchanged.

Return *measure*.

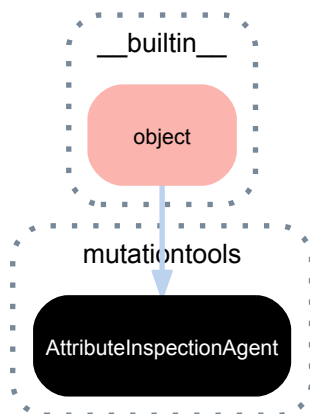




# MUTATIONTOOLS

## 17.1 Concrete classes

### 17.1.1 mutationtools.AttributeInspectionAgent



**class** `mutationtools.AttributeInspectionAgent` (*component*)  
Inspect one component.

#### Bases

- `__builtin__.object`

#### Methods

`AttributeInspectionAgent.get_annotation` (*name*, *default=None*)

Gets value of annotation with *name* attached to component.

Returns *default* when no annotation with *name* is attached to component.

Raises exception when more than one annotation with *name* is attached to component.

`AttributeInspectionAgent.get_badly_formed_components` ()

Gets badly formed components in component.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff[1].written_duration = Duration(1, 4)
>>> spannertools.BeamSpanner(staff[:])
BeamSpanner(c'8, d'4, e'8, f'8)
```

```
>>> inspect(staff).get_badly_formed_components()
[Note("d'4")]
```

(Beamed quarter notes are not well formed.)

Returns list.

`AttributeInspectionAgent.get_components` (*component\_classes=None*, *include\_self=True*) *in-*

Gets all components of *component\_classes* in the descendants of component.

Returns component selection.

`AttributeInspectionAgent.get_contents` (*include\_self=True*)

Gets contents of component.

Returns sequential selection.

`AttributeInspectionAgent.get_descendants` (*include\_self=True*)

Gets descendants of component.

Returns descendants.

`AttributeInspectionAgent.get_duration` (*in\_seconds=False*)

Gets duration of component.

Returns duration.

`AttributeInspectionAgent.get_effective_context_mark` (*context\_mark\_classes=None*)

Gets effective context mark of *context\_mark\_class* that governs component.

Returns context mark or none.

`AttributeInspectionAgent.get_effective_staff` ()

Gets effective staff of component.

Returns staff or none.

`AttributeInspectionAgent.get_grace_containers` (*kind=None*)

Gets grace containers attached to leaf.

**Example 1.** Get all grace containers attached to note:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> grace_container = containertools.GraceContainer(
...     [Note("cs'16")],
...     kind='grace',
... )
>>> grace_container.attach(staff[1])
Note("d'8")
>>> after_grace = containertools.GraceContainer(
...     [Note("ds'16")],
...     kind='after'
... )
>>> after_grace.attach(staff[1])
Note("d'8")
>>> show(staff)
```



```
>>> inspect(staff[1]).get_grace_containers()
(GraceContainer(cs'16), GraceContainer(ds'16))
```

**Example 2.** Get only (proper) grace containers attached to note:

```
>>> inspect(staff[1]).get_grace_containers(kind='grace')
(GraceContainer(cs'16),)
```

**Example 3.** Get only after grace containers attached to note:

```
>>> inspect(staff[1]).get_grace_containers(kind='after')
(GraceContainer(ds'16),)
```

Set *kind* to 'grace', 'after' or none.

Returns tuple.

`AttributeInspectionAgent.get_leaf(n=0)`

Gets leaf *n* in logical voice.

```
>>> staff = Staff()
>>> staff.append(Voice("c'8 d'8 e'8 f'8"))
>>> staff.append(Voice("g'8 a'8 b'8 c''8"))
>>> show(staff)
```



```
>>> for n in range(8):
...     print n, inspect(staff[0][0]).get_leaf(n)
...
0 c'8
1 d'8
2 e'8
3 f'8
4 None
5 None
6 None
7 None
```

Returns leaf or none.

`AttributeInspectionAgent.get_lineage()`

Gets lineage of component.

Returns lineage.

`AttributeInspectionAgent.get_mark(mark_classes=None)`

Gets exactly one mark of *mark\_classes* attached to component.

Raises exception when no mark of *mark\_classes* is attached to component.

Returns mark.

`AttributeInspectionAgent.get_marks(mark_classes=None)`

Get all marks of *mark\_classes* attached to component.

Returns tuple.

`AttributeInspectionAgent.get_markup(direction=None)`

Gets all markup attached to component.

Returns tuple.

`AttributeInspectionAgent.get_parentage(include_self=True)`

Gets parentage of component.

Returns parentage.

`AttributeInspectionAgent.get_spanner(spanner_classes=None)`

Gets exactly one spanner of *spanner\_classes* attached to component.

Raises exception when no spanner of *spanner\_classes* is attached to component.

Returns spanner.

`AttributeInspectionAgent.get_spanners(spanner_classes=None)`

Gets spanners attached to component.

Returns set.

`AttributeInspectionAgent.get_tie_chain()`

Gets tie chain that governs leaf.

Returns tie chain.

`AttributeInspectionAgent.get_timespan (in_seconds=False)`

Gets timespan of component.

Returns timespan.

`AttributeInspectionAgent.get_vertical_moment (governor=None)`

Gets vertical moment starting with component.

Returns vertical moment.

`AttributeInspectionAgent.get_vertical_moment_at (offset)`

Gets vertical moment at *offset*.

Returns vertical moment.

`AttributeInspectionAgent.is_bar_line_crossing()`

True when component crosses bar line. Otherwise false.

**Example.**

```
>>> staff = Staff("c'4 d'4 e'4")
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8)) (Staff{3})
>>> show(staff)
```



```
>>> for note in staff:
...     result = inspect(note).is_bar_line_crossing()
...     print note, result
...
c'4 False
d'4 True
e'4 False
```

Returns boolean.

`AttributeInspectionAgent.is_well_formed (allow_empty_containers=True)`

True when component is well-formed. Otherwise false.

Returns false.

`AttributeInspectionAgent.report_modifications()`

Reports modifications of component.

**Example.** Report modifications of container in selection:

```
>>> container = Container("c'8 d'8 e'8 f'8")
>>> container.override.note_head.color = 'red'
>>> container.override.note_head.style = 'harmonic'
>>> show(container)
```



```
>>> report = inspect(container).report_modifications()
```

```
>>> print report
{
  \override NoteHead #'color = #red
  \override NoteHead #'style = #'harmonic
  %% 4 components omitted %%
  \revert NoteHead #'color
  \revert NoteHead #'style
}
```

Returns string.

`AttributeInspectionAgent.tabulate_well_formedness_violations()`

Tabulates well-formedness violations in component.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff[1].written_duration = Duration(1, 4)
>>> spannertools.BeamSpanner(staff[:])
BeamSpanner(c'8, d'4, e'8, f'8)
```

```
>>> result = inspect(staff).tabulate_well_formedness_violations()
```

```
>>> print result
1 /      4 beamed quarter note
0 /      1 discontinuous spanner
0 /      5 duplicate id
0 /      1 empty container
0 /      0 intermarked hairpin
0 /      0 misdurated measure
0 /      0 misfilled measure
0 /      4 mispitched tie
0 /      4 misrepresented flag
0 /      5 missing parent
0 /      0 nested measure
0 /      1 overlapping beam
0 /      0 overlapping glissando
0 /      0 overlapping octavation
0 /      0 short hairpin
```

Beamed quarter notes are not well formed.

Returns string.

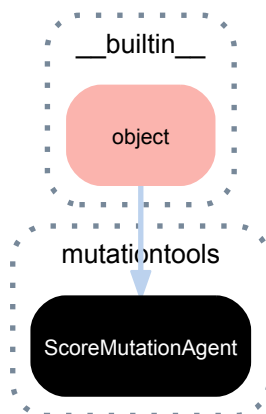
## Special methods

`AttributeInspectionAgent.__repr__()`

Interpreter representation of attribute inspection agent.

Returns string.

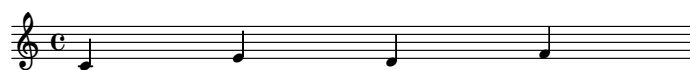
## 17.1.2 mutationtools.ScoreMutationAgent



**class** `mutationtools.ScoreMutationAgent` (*client*)

A wrapper around the Abjad score mutators.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



```
>>> mutate(staff[2:])
ScoreMutationAgent(SliceSelection(Note("d'4"), Note("f'4")))
```

## Bases

- `__builtin__.object`

## Methods

`ScoreMutationAgent.copy` (*n=1, include\_enclosing\_containers=False*)

Copies component and fractures crossing spanners.

Returns new component.

`ScoreMutationAgent.extract` (*scale\_contents=False*)

Extracts mutation client from score.

Leaves children of mutation client in score.

**Example 1.** Extract tuplet:

```
>>> staff = Staff()
>>> time_signature = contexttools.TimeSignatureMark((3, 4))
>>> time_signature = time_signature.attach(staff)
>>> staff.append(Tuplet((3, 2), "c'4 e'4"))
>>> staff.append(Tuplet((3, 2), "d'4 f'4"))
>>> hairpin = spannertools.HairpinSpanner([], 'p < f')
>>> hairpin.attach(staff.select_leaves())
>>> show(staff)
```



```
>>> empty_tuplet = mutate(staff[-1]).extract()
>>> empty_tuplet = mutate(staff[0]).extract()
>>> show(staff)
```



**Example 2.** Scale tuplet contents and then extract tuplet:

```
>>> staff = Staff()
>>> time_signature = contexttools.TimeSignatureMark((3, 4))
>>> time_signature = time_signature.attach(staff)
>>> staff.append(Tuplet((3, 2), "c'4 e'4"))
>>> staff.append(Tuplet((3, 2), "d'4 f'4"))
>>> hairpin = spannertools.HairpinSpanner([], 'p < f')
>>> hairpin.attach(staff.select_leaves())
>>> show(staff)
```



```
>>> empty_tuplet = mutate(staff[-1]).extract(
...     scale_contents=True)
>>> empty_tuplet = mutate(staff[0]).extract(
...     scale_contents=True)
>>> show(staff)
```



Returns mutation client.

`ScoreMutationAgent.fuse()`

Fuses mutation client.

**Example 1.** Fuse in-score leaves:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> mutate(staff[1:]).fuse()
[Note("d'4.") ]
>>> show(staff)
```



**Example 2.** Fuse parent-contiguous fixed-duration tuplets in selection:

```
>>> tuplet_1 = tuplettools.FixedDurationTuplet(
...     Duration(2, 8), [])
>>> tuplet_1.extend("c'8 d'8 e'8")
>>> beam = spannertools.BeamSpanner(tuplet_1[:])
>>> tuplet_2 = tuplettools.FixedDurationTuplet(
...     Duration(2, 16), [])
>>> tuplet_2.extend("c'16 d'16 e'16")
>>> slur = spannertools.SlurSpanner(tuplet_2[:])
>>> staff = Staff([tuplet_1, tuplet_2])
>>> show(staff)
```



```
>>> tuplets = staff[:]
>>> mutate(tuplets).fuse()
FixedDurationTuplet(3/8, [c'8, d'8, e'8, c'16, d'16, e'16])
>>> show(staff)
```



Returns new tuplet.

Fuses zero or more parent-contiguous *tuplets*.

Allows in-score *tuplets*.

Allows outside-of-score *tuplets*.

All *tuplets* must carry the same multiplier.

All *tuplets* must be of the same type.

**Example 3.** Fuse in-score measures:

```
>>> staff = Staff()
>>> staff.append(Measure((1, 4), "c'8 d'8"))
>>> staff.append(Measure((2, 8), "e'8 f'8"))
>>> slur = spannertools.SlurSpanner()
>>> slur = slur.attach(staff[:])
>>> show(staff)
```



```
>>> measures = staff[:]
>>> mutate(measures).fuse()
Measure(2/4, [c'8, d'8, e'8, f'8])
>>> show(staff)
```



Returns fused mutation client.

`ScoreMutationAgent.replace(recipients)`

Replaces mutation client (and contents of mutation client) with *recipients*.

**Example 1.** Replace in-score tuplet (and children of tuplet) with notes. Functions exactly the same as container `setitem`:

```
>>> tuplet_1 = Tuplet((2, 3), "c'4 d'4 e'4")
>>> tuplet_2 = Tuplet((2, 3), "d'4 e'4 f'4")
>>> staff = Staff([tuplet_1, tuplet_2])
>>> hairpin = spannertools.HairpinSpanner([], 'p < f')
>>> hairpin.attach(staff[:])
>>> slur = spannertools.SlurSpanner()
>>> slur.attach(staff.select_leaves())
>>> show(staff)
```



```
>>> notes = notetools.make_notes(
...     "c' d' e' f' c' d' e' f'",
...     Duration(1, 16),
... )
>>> mutate([tuplet_1]).replace(notes)
>>> show(staff)
```



Preserves both hairpin and slur.

Returns none.

`ScoreMutationAgent.respell_with_flats()`

Respell named pitches in mutation client with flats:

```
>>> staff = Staff("c'8 cs'8 d'8 ef'8 e'8 f'8")
>>> show(staff)
```



```
>>> mutate(staff).respell_with_flats()
>>> show(staff)
```



Returns none.

`ScoreMutationAgent.respell_with_sharps()`

Respell named pitches in mutation client with sharps:

```
>>> staff = Staff("c'8 cs'8 d'8 ef'8 e'8 f'8")
>>> show(staff)
```





```
>>> mutate(staff).respell_with_sharps()
>>> show(staff)
```



Returns none.

ScoreMutationAgent.**scale** (*multiplier*)

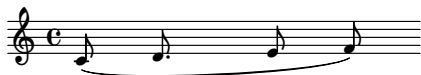
Scales mutation client by *multiplier*.

**Example 1a.** Scale note duration by dot-generating multiplier:

```
>>> staff = Staff("c'8 ( d'8 e'8 f'8 )")
>>> show(staff)
```



```
>>> mutate(staff[1]).scale(Multiplier(3, 2))
>>> show(staff)
```

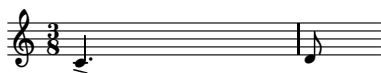


**Example 1b.** Scale nontrivial tie chain by dot-generating *multiplier*:

```
>>> staff = Staff(r"c'8 \accent ~ c'8 d'8")
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> time_signature = time_signature.attach(staff)
>>> show(staff)
```



```
>>> tie_chain = inspect(staff[0]).get_tie_chain()
>>> tie_chain = mutate(tie_chain).scale(Multiplier(3, 2))
>>> show(staff)
```



**Example 1c.** Scale container by dot-generating multiplier:

```
>>> container = Container(r"c'8 ( d'8 e'8 f'8 )")
>>> show(container)
```



```
>>> mutate(container).scale(Multiplier(3, 2))
>>> show(container)
```



**Example 2a.** Scale note by tie-generating multiplier:

```
>>> staff = Staff("c'8 ( d'8 e'8 f'8 )")
>>> show(staff)
```



```
>>> mutate(staff[1]).scale(Multiplier(5, 4))
>>> show(staff)
```



**Example 2b.** Scale nontrivial tie chain by tie-generating *multiplier*:

```
>>> staff = Staff(r"c'8 \accent ~ c'8 d'16")
>>> time_signature = contexttools.TimeSignatureMark((5, 16))
>>> time_signature.attach(staff)
>>> show(staff)
```



```
>>> tie_chain = inspect(staff[0]).get_tie_chain()
>>> tie_chain = mutate(tie_chain).scale(Multiplier(5, 4))
>>> show(staff)
```



**Example 2c.** Scale container by tie-generating multiplier:

```
>>> container = Container(r"c'8 ( d'8 e'8 f'8 )")
>>> show(container)
```



```
>>> mutate(container).scale(Multiplier(5, 4))
>>> show(container)
```



**Example 3a.** Scale note by tuplet-generating multiplier:

```
>>> staff = Staff(r"c'8 ( d'8 e'8 f'8 )")
>>> show(staff)
```



```
>>> mutate(staff[1]).scale(Multiplier(2, 3))
>>> show(staff)
```



**Example 3b.** Scale trivial tie chain by tuplet-generating multiplier:

```
>>> staff = Staff(r"c'8 \accent")
>>> show(staff)
```



```
>>> tie_chain = inspect(staff[0]).get_tie_chain()
>>> tie_chain = mutate(tie_chain).scale(Multiplier(4, 3))
>>> show(staff)
```

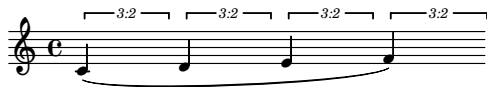


**Example 3c.** Scale container by tuplet-generating multiplier:

```
>>> container = Container(r"c'8 ( d'8 e'8 f'8 )")
>>> show(container)
```



```
>>> mutate(container).scale(Multiplier(4, 3))
>>> show(container)
```



**Example 4.** Scale note by tie- and tuplet-generating multiplier:

```
>>> staff = Staff("c'8 ( d'8 e'8 f'8 )")
>>> show(staff)
```



```
>>> mutate(staff[1]).scale(Multiplier(5, 6))
>>> show(staff)
```



**Example 5.** Scale note carrying LilyPond multiplier:

```
>>> note = Note("c'8")
>>> note.lilypond_duration_multiplier = Duration(1, 2)
>>> show(note)
```



```
>>> mutate(note).scale(Multiplier(5, 3))
>>> show(note)
```



**Example 6.** Scale tuplet:

```
>>> staff = Staff()
>>> time_signature = contexttools.TimeSignatureMark((4, 8))
>>> time_signature = time_signature.attach(staff)
>>> tuplet = tuplettools.Tuplet((4, 5), [])
>>> tuplet.extend("c'8 d'8 e'8 f'8 g'8")
>>> staff.append(tuplet)
>>> show(staff)
```



```
>>> mutate(tuplet).scale(Multiplier(2))
>>> show(staff)
```



**Example 7.** Scale fixed-duration tuplet:

```
>>> staff = Staff()
>>> time_signature = contexttools.TimeSignatureMark((4, 8))
>>> time_signature = time_signature.attach(staff)
>>> tuplet = tuplettools.FixedDurationTuplet((4, 8), [])
>>> tuplet.extend("c'8 d'8 e'8 f'8 g'8")
>>> staff.append(tuplet)
>>> show(staff)
```



```
>>> mutate(tuplet).scale(Multiplier(2))
>>> show(staff)
```



Returns none.

`ScoreMutationAgent.splice` (*components*, *direction=Right*, *grow\_spanners=True*)

Splices *components* to the right or left of selection.

Returns list of components.

`ScoreMutationAgent.split` (*durations*, *fracture\_spanners=False*, *cyclic=False*,  
*tie\_split\_notes=True*)

Splits component or selection by *durations*.

**Example 1.** Split leaves:

```
>>> staff = Staff("c'8 e' d' f' c' e' d' f'")
>>> leaves = staff.select_leaves()
>>> spanner = spannertools.HairpinSpanner(leaves, 'p < f')
>>> staff.override.dynamic_line_spanner.staff_padding = 3
>>> show(staff)
```



```
>>> durations = [Duration(3, 16), Duration(7, 32)]
>>> result = mutate(leaves).split(
...     durations,
...     tie_split_notes=False,
... )
>>> show(staff)
```



**Example 2.** Split leaves and fracture crossing spanners:

```
>>> staff = Staff("c'8 e' d' f' c' e' d' f'")
>>> leaves = staff.select_leaves()
>>> spanner = spannertools.HairpinSpanner(leaves, 'p < f')
>>> staff.override.dynamic_line_spanner.staff_padding = 3
>>> show(staff)
```



```
>>> durations = [Duration(3, 16), Duration(7, 32)]
>>> result = mutate(leaves).split(
...     durations,
...     fracture_spanners=True,
...     tie_split_notes=False,
... )
>>> show(staff)
```



**Example 3.** Split leaves cyclically:

```
>>> staff = Staff("c'8 e' d' f' c' e' d' f'")
>>> leaves = staff.select_leaves()
>>> spanner = spannertools.HairpinSpanner(leaves, 'p < f')
>>> staff.override.dynamic_line_spanner.staff_padding = 3
>>> show(staff)
```



```
>>> durations = [Duration(3, 16), Duration(7, 32)]
>>> result = mutate(leaves).split(
...     durations,
...     cyclic=True,
...     tie_split_notes=False,
... )
>>> show(staff)
```



**Example 4.** Split leaves cyclically and fracture spanners:

```
>>> staff = Staff("c'8 e' d' f' c' e' d' f'")
>>> leaves = staff.select_leaves()
>>> spanner = spannertools.HairpinSpanner(leaves, 'p < f')
>>> staff.override.dynamic_line_spanner.staff_padding = 3
>>> show(staff)
```



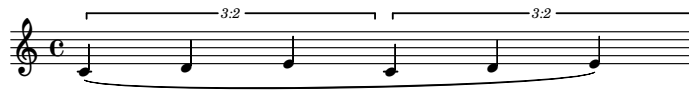
```
>>> durations = [Duration(3, 16), Duration(7, 32)]
>>> result = mutate(leaves).split(
...     durations,
...     cyclic=True,
...     fracture_spanners=True,
...     tie_split_notes=False,
... )
>>> show(staff)
```



**Example 5.** Split tupletted leaves and fracture crossing spanners:

```
>>> staff = Staff()
>>> staff.append(Tuplet((2, 3), "c'4 d' e'"))
>>> staff.append(Tuplet((2, 3), "c'4 d' e'"))
```

```
>>> leaves = staff.select_leaves()
>>> spanner = spannertools.SlurSpanner(leaves)
>>> show(staff)
```



```
>>> durations = [Duration(1, 4)]
>>> result = mutate(leaves).split(
...     durations,
...     fracture_spanners=True,
...     tie_split_notes=False,
... )
>>> show(staff)
```



Returns list of selections.

`ScoreMutationAgent.swap(container)`  
Swaps mutation client for empty *container*.

**Example 1.** Swap measures for tuplet:

```
>>> staff = Staff()
>>> staff.append(Measure((3, 4), "c'4 d'4 e'4"))
>>> staff.append(Measure((3, 4), "d'4 e'4 f'4"))
>>> leaves = staff.select_leaves()
>>> hairpin = spannertools.HairpinSpanner([], 'p < f')
>>> hairpin.attach(leaves)
>>> measures = staff[:]
>>> slur = spannertools.SlurSpanner()
>>> slur.attach(measures)
>>> show(staff)
```



```
>>> measures = staff[:]
>>> tuplet = Tuplet(Multiplier(2, 3), [])
>>> tuplet.preferred_denominator = 4
>>> mutate(measures).swap(tuplet)
>>> show(staff)
```

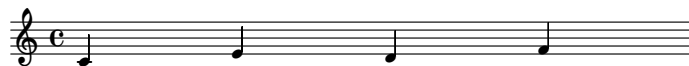


Returns none.

## Special methods

`ScoreMutationAgent.__repr__()`  
Interpreter representation of score mutation agent.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



```
>>> mutate(staff[2:])  
ScoreMutationAgent(SliceSelection(Note("d'4"), Note("f'4")))
```

Returns string.

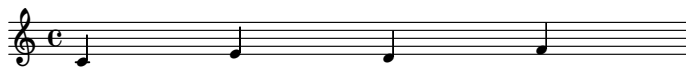
## 17.2 Functions

### 17.2.1 mutationtools.mutate

`mutationtools.mutate(expr)`

Mutate *expr*.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")  
>>> show(staff)
```



```
>>> notes = staff[-2:]  
>>> mutate(notes)  
ScoreMutationAgent(SliceSelection(Note("d'4"), Note("f'4")))
```

Returns score mutation agent.

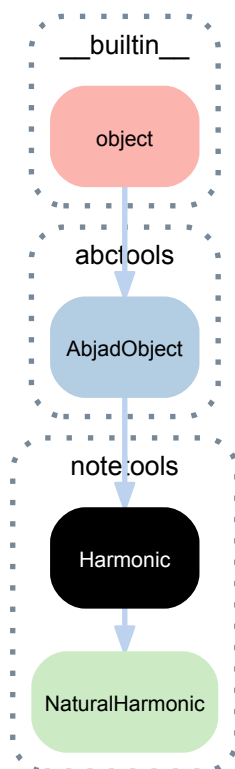




# NOTETOOLS

## 18.1 Concrete classes

### 18.1.1 notetools.Harmonic



**class** `notetools.Harmonic`

Abjad model of both natural and artificial harmonics. Abstract base class.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

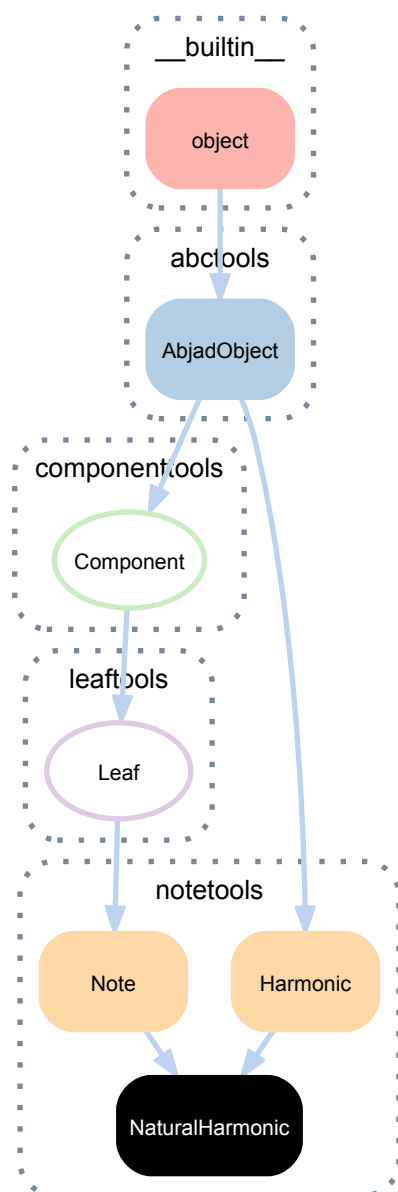
`Harmonic.suono_reale`

Actual sound of the harmonic when played.

## Special methods

- (AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.
- (AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.
- (AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 18.1.2 notetools.NaturalHarmonic



**class** notetools.**NaturalHarmonic**(\*args)  
 Abjad model of natural harmonic.

Initialize natural harmonic by hand:

```
>>> notetools.NaturalHarmonic("cs'8.")
NaturalHarmonic(cs', 8.)
```

Initialize natural harmonic from note:

```
>>> note = Note("cs'8.")
```

```
>>> notetools.NaturalHarmonic(note)
NaturalHarmonic(cs', 8.)
```

Natural harmonics are immutable.

## Bases

- `notetools.Note`
- `leaftools.Leaf`
- `componenttools.Component`
- `notetools.Harmonic`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Component).lilypond_format`

Lilypond format of component.

Returns string.

`(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

`(Component).storage_format`

Storage format of component.

Returns string.

`(Harmonic).suono_reale`

Actual sound of the harmonic when played.

## Read/write properties

`(Leaf).lilypond_duration_multiplier`

LilyPond duration multiplier.

Set to positive multiplier or none.

Returns positive multiplier or none.

`(Note).note_head`

Get note head of note:

```
>>> note = Note(13, (3, 16))
>>> note.note_head
NoteHead("cs'")
```

Set note head of note:

```
>>> note = Note(13, (3, 16))
>>> note.note_head = 14
>>> note
Note("d'8.")
```

(Note) **.sounding\_pitch**

Get sounding pitch of note:

```
>>> staff = Staff("d'8 e'8 f'8 g'8")
>>> piccolo = instrumenttools.Piccolo()(staff)
```

```
>>> instrumenttools.transpose_from_sounding_pitch_to_written_pitch(
...     staff)
```

Set sounding pitch of note:

```
>>> staff[0].sounding_pitch = "dqs'"
>>> f(staff)
\new Staff {
  \set Staff.instrumentName = \markup { Piccolo }
  \set Staff.shortInstrumentName = \markup { Picc. }
  dqs'8
  e'8
  f'8
  g'8
}
```

(Leaf) **.written\_duration**

Written duration of leaf.

Set to duration.

Returns duration.

(Note) **.written\_pitch**

Get named pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch
NamedPitch("cs'")
```

Set named pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch = 14
>>> note
Note("d'8.")
```

(Leaf) **.written\_pitch\_indication\_is\_at\_sounding\_pitch**

Returns true when written pitch is at sounding pitch. Returns false when written pitch is transposed.

(Leaf) **.written\_pitch\_indication\_is\_nonsemantic**

Returns true when pitch is nonsemantic. Returns false otherwise.

Set to true when using leaves only graphically.

Setting this value to true sets sounding pitch indicator to false.

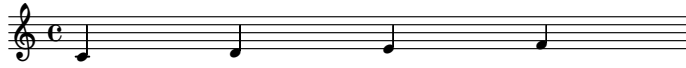
## Methods

(Note) **.add\_artificial\_harmonic** (*named\_interval=None*)

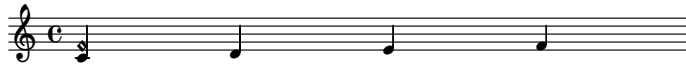
Adds artifical harmonic to note at *named\_interval*.

**Example.** Add artificial harmonic to note at the perfect fourth above.

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> spannertools.BeamSpanner(staff[:])
BeamSpanner(c'4, d'4, e'4, f'4)
>>> show(staff)
```



```
>>> staff[0].add_artificial_harmonic()
Chord("<c' f'>4")
>>> show(staff)
```



Sets *named\_interval* to a perfect fourth above when *named\_interval=None*.

Creates new chord from *note*.

Moves parentage and spanners from *note* to chord.

Returns chord.

(Component) .**select** (*sequential=False*)

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

## Special methods

(Component) .**\_\_copy\_\_** (\*args)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(AbjadObject) .**\_\_eq\_\_** (expr)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Component) .**\_\_mul\_\_** (n)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject) .**\_\_ne\_\_** (expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

NaturalHarmonic .**\_\_repr\_\_** ()

(Component) .**\_\_rmul\_\_** (n)

Copies component *n* times and detach spanners.

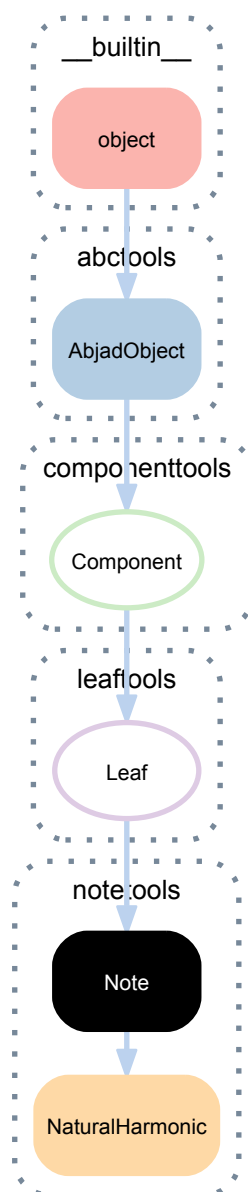
Returns list of new components.

(Leaf) .**\_\_str\_\_** ()

String representation of leaf.

Returns string.

### 18.1.3 notetools.Note



**class** `notetools.Note` (\*args, \*\*kwargs)  
A note.

**Example.**

```
>>> note = Note("cs'8.")
>>> measure = Measure((3, 16), [note])
>>> show(measure)
```



#### Bases

- `leaftools.Leaf`
- `componenttools.Component`
- `abctools.AbjadObject`

- `__builtin__.object`

## Read-only properties

`(Component).lilypond_format`

Lilypond format of component.

Returns string.

`(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

`(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

`(Leaf).lilypond_duration_multiplier`

LilyPond duration multiplier.

Set to positive multiplier or none.

Returns positive multiplier or none.

`Note.note_head`

Get note head of note:

```
>>> note = Note(13, (3, 16))
>>> note.note_head
NoteHead("cs'")
```

Set note head of note:

```
>>> note = Note(13, (3, 16))
>>> note.note_head = 14
>>> note
Note("d'8.")
```

`Note.sounding_pitch`

Get sounding pitch of note:

```
>>> staff = Staff("d'8 e'8 f'8 g'8")
>>> piccolo = instrumenttools.Piccolo()(staff)
```

```
>>> instrumenttools.transpose_from_sounding_pitch_to_written_pitch(
...     staff)
```

Set sounding pitch of note:

```
>>> staff[0].sounding_pitch = "dqs'"
>>> f(staff)
\new Staff {
  \set Staff.instrumentName = \markup { Piccolo }
  \set Staff.shortInstrumentName = \markup { Picc. }
  dqs'8
  e'8
  f'8
  g'8
}
```

(Leaf).**written\_duration**

Written duration of leaf.

Set to duration.

Returns duration.

Note.**written\_pitch**

Get named pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch
NamedPitch("cs' ")
```

Set named pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch = 14
>>> note
Note("d''8.")
```

(Leaf).**written\_pitch\_indication\_is\_at\_sounding\_pitch**

Returns true when written pitch is at sounding pitch. Returns false when written pitch is transposed.

(Leaf).**written\_pitch\_indication\_is\_nonsemantic**

Returns true when pitch is nonsemantic. Returns false otherwise.

Set to true when using leaves only graphically.

Setting this value to true sets sounding pitch indicator to false.

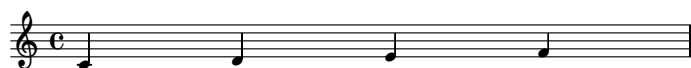
## Methods

Note.**add\_artificial\_harmonic** (*named\_interval=None*)

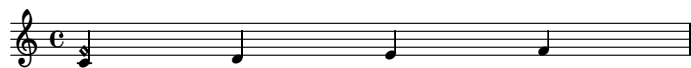
Adds artifical harmonic to note at *named\_interval*.

**Example.** Add artificial harmonic to note at the perfect fourth above.

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> spannertools.BeamSpanner(staff[:])
BeamSpanner(c'4, d'4, e'4, f'4)
>>> show(staff)
```



```
>>> staff[0].add_artificial_harmonic()
Chord("<c' f'>4")
>>> show(staff)
```



Sets *named\_interval* to a perfect fourth above when *named\_interval=None*.

Creates new chord from *note*.

Moves parentage and spanners from *note* to chord.

Returns chord.

(Component).**select** (*sequential=False*)

Selects component.

Returns component selection when *sequential* is false.

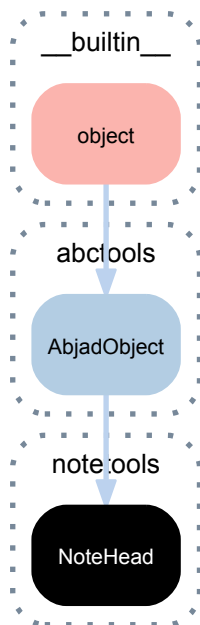
Returns sequential selection when *sequential* is true.



## Special methods

- (Component) .**\_\_copy\_\_** (\*args)  
Copies component with marks but without children of component or spanners attached to component.  
Returns new component.
- (AbjadObject) .**\_\_eq\_\_** (expr)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.
- (Component) .**\_\_mul\_\_** (n)  
Copies component *n* times and detaches spanners.  
Returns list of new components.
- (AbjadObject) .**\_\_ne\_\_** (expr)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.
- (Leaf) .**\_\_repr\_\_** ()  
Interpreter representation of leaf.  
Returns string.
- (Component) .**\_\_rmul\_\_** (n)  
Copies component *n* times and detach spanners.  
Returns list of new components.
- (Leaf) .**\_\_str\_\_** ()  
String representation of leaf.  
Returns string.

### 18.1.4 notetools.NoteHead



```
class notetools.NoteHead (written_pitch=None, client=None, is_cautionary=False,
                           is_forced=False, tweak_pairs=())
    Abjad model of a note head:
```

```
>>> notetools.NoteHead(13)
NoteHead("cs' ' ")
```

Note heads are immutable.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

**NoteHead.lilypond\_format**

LilyPond input format of note head:

```
>>> note_head = notetools.NoteHead("cs' ' ")
>>> note_head.lilypond_format
"cs' ' "
```

Returns string.

**NoteHead.named\_pitch**

Named pitch equal to note head:

```
>>> note_head = notetools.NoteHead("cs' ' ")
>>> note_head.named_pitch
NamedPitch("cs' ' ")
```

Returns named pitch.

**NoteHead.tweak**

LilyPond tweak reservoir:

```
>>> note_head = notetools.NoteHead("cs' ' ")
>>> note_head.tweak
LilyPondTweakReservoir()
```

Returns LilyPond tweak reservoir.

## Read/write properties

**NoteHead.is\_cautionary**

Get cautionary accidental flag:

```
>>> note_head = notetools.NoteHead("cs' ' ")
>>> note_head.is_cautionary
False
```

Set cautionary accidental flag:

```
>>> note_head = notetools.NoteHead("cs' ' ")
>>> note_head.is_cautionary = True
```

Returns boolean.

**NoteHead.is\_forced**

**NoteHead.written\_pitch**

Get named pitch of note head:

```
>>> note_head = notetools.NoteHead("cs' ' ")
>>> note_head.written_pitch
NamedPitch("cs' ' ")
```

Set named pitch of note head:

```
>>> note_head = notetools.NoteHead("cs'")
>>> note_head.written_pitch = "d'"
>>> note_head.written_pitch
NamedPitch("d'")
```

Set pitch token.

## Special methods

`NoteHead.__copy__(*args)`

`NoteHead.__eq__(expr)`

`NoteHead.__ge__(other)`

`x.__ge__(y) <==> x>=y`

`NoteHead.__gt__(other)`

`x.__gt__(y) <==> x>y`

`NoteHead.__le__(other)`

`x.__le__(y) <==> x<=y`

`NoteHead.__lt__(expr)`

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`NoteHead.__repr__()`

`NoteHead.__str__()`

## 18.2 Functions

### 18.2.1 `notetools.make_accelerating_notes_with_lilypond_multipliers`

`notetools.make_accelerating_notes_with_lilypond_multipliers(pitches, total, start, stop, exp='cosine', written=None)`

Make accelerating notes with LilyPond multipliers:

```
>>> pitches = ['C#4', 'D4']
>>> total = Duration(4, 4)
>>> start = Duration(1, 4)
>>> stop = Duration(1, 16)
>>> args = [pitches, total, start, stop]
```

```
>>> notes = notetools.make_accelerating_notes_with_lilypond_multipliers(*args)
```

```
>>> staff = Staff(notes)
>>> beam = spannertools.BeamSpanner(staff[:])
>>> slur = spannertools.SlurSpanner(staff[:])
```

```
>>> show(staff)
```



Set note pitches cyclically from *pitches*.

Returns as many interpolation values as necessary to fill the *total* duration requested.

Interpolate durations from *start* to *stop*.

Set note durations to *written* duration times computed interpolated multipliers.

Interpret *written=None* as eighth notes.

Returns list of notes.

## 18.2.2 notetools.make\_notes

`notetools.make_notes(pitches, durations, decrease_durations_monotonically=True)`

Make notes according to *pitches* and *durations*.

Cycle through *pitches* when the length of *pitches* is less than the length of *durations*:

```
>>> notetools.make_notes([0], [(1, 16), (1, 8), (1, 8)])
Selection(Note("c'16"), Note("c'8"), Note("c'8"))
```

Cycle through *durations* when the length of *durations* is less than the length of *pitches*:

```
>>> notetools.make_notes([0, 2, 4, 5, 7], [(1, 16), (1, 8), (1, 8)])
Selection(Note("c'16"), Note("d'8"), Note("e'8"), Note("f'16"), Note("g'8"))
```

Create ad hoc tuplets for nonassignable durations:

```
>>> notetools.make_notes([0], [(1, 16), (1, 12), (1, 8)])
Selection(Note("c'16"), Tuplet(2/3, [c'8]), Note("c'8"))
```

Set `decrease_durations_monotonically=True` to express tied values in decreasing duration:

```
>>> notetools.make_notes(
...     [0],
...     [(13, 16)],
...     decrease_durations_monotonically=True,
... )
Selection(Note("c'2."), Note("c'16"))
```

Set `decrease_durations_monotonically=False` to express tied values in increasing duration:

```
>>> notetools.make_notes(
...     [0],
...     [(13, 16)],
...     decrease_durations_monotonically=False,
... )
Selection(Note("c'16"), Note("c'2.))
```

Set *pitches* to a single pitch or a sequence of pitches.

Set *durations* to a single duration or a list of durations.

Returns list of newly constructed notes.

## 18.2.3 notetools.make\_notes\_with\_multiplied\_durations

`notetools.make_notes_with_multiplied_durations(pitch, written_duration, multiplied_durations)`

Make *written\_duration* notes with *pitch* and *multiplied\_durations*:

```
>>> args = [0, Duration(1, 4), [(1, 2), (1, 3), (1, 4), (1, 5)]]
>>> notetools.make_notes_with_multiplied_durations(*args)
Selection(Note("c'4 * 2"), Note("c'4 * 4/3"), Note("c'4 * 1"), Note("c'4 * 4/5"))
```

Useful for making spatially positioned notes.

Returns list of notes.

### 18.2.4 notetools.make\_percussion\_note

`notetools.make_percussion_note` (*pitch*, *total\_duration*, *max\_note\_duration*=(1, 8))

Makes short note with *max\_note\_duration* followed by rests together totaling *total\_duration*.

```
>>> leaves = notetools.make_percussion_note(2, (1, 4), (1, 8))
>>> staff = Staff(leaves)
>>> show(staff)
```



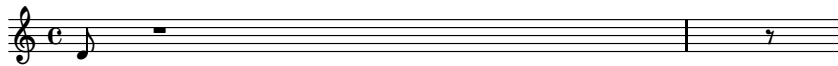
```
>>> leaves = notetools.make_percussion_note(2, (1, 64), (1, 8))
>>> staff = Staff(leaves)
>>> show(staff)
```



```
>>> leaves = notetools.make_percussion_note(2, (5, 64), (1, 8))
>>> staff = Staff(leaves)
>>> show(staff)
```



```
>>> leaves = notetools.make_percussion_note(2, (5, 4), (1, 8))
>>> staff = Staff(leaves)
>>> show(staff)
```



Returns list of newly constructed note followed by zero or more newly constructed rests.

Durations of note and rests returned will sum to *total\_duration*.

Duration of note returned will be no greater than *max\_note\_duration*.

Duration of rests returned will sum to note duration taken from *total\_duration*.

Useful for percussion music where attack duration is negligible and tied notes undesirable.

### 18.2.5 notetools.make\_quarter\_notes\_with\_lilypond\_duration\_multiplier

`notetools.make_quarter_notes_with_lilypond_duration_multiplier` (*pitches*,  
*multiplied\_durations*)

Make quarter notes with *pitches* and *multiplied\_durations*:

```
>>> args = [[0, 2, 4, 5], [(1, 4), (1, 5), (1, 6), (1, 7)]]
>>> notetools.make_quarter_notes_with_lilypond_duration_multiplier(*args)
Selection(Note("c'4 * 1"), Note("d'4 * 4/5"), Note("e'4 * 2/3"), Note("f'4 * 4/7"))
```

Read *pitches* cyclically where the length of *pitches* is less than the length of *multiplied\_durations*:

```
>>> args = [[0], [(1, 4), (1, 5), (1, 6), (1, 7)]]
>>> notetools.make_quarter_notes_with_lilypond_duration_multiplier(*args)
Selection(Note("c'4 * 1"), Note("c'4 * 4/5"), Note("c'4 * 2/3"), Note("c'4 * 4/7"))
```

Read *multiplied\_durations* cyclically where the length of *multiplied\_durations* is less than the length of *pitches*:

```
>>> args = [[0, 2, 4, 5], [(1, 5)]]
>>> notetools.make_quarter_notes_with_lilypond_duration_multiplier(*args)
Selection(Note("c'4 * 4/5"), Note("d'4 * 4/5"), Note("e'4 * 4/5"),
Note("f'4 * 4/5"))
```

Returns list of zero or more newly constructed notes.

### 18.2.6 `notetools.make_repeated_notes`

`notetools.make_repeated_notes` (*count*, *duration*=*Duration*(1, 8))

Make *count* repeated notes with note head-assignable *duration*:

```
>>> notetools.make_repeated_notes(4)
Selection(Note("c'8"), Note("c'8"), Note("c'8"), Note("c'8"))
```

Make *count* repeated tie chains with tied *duration*:

```
>>> notes = notetools.make_repeated_notes(2, (5, 16))
>>> voice = Voice(notes)
```

Make ad hoc tuplet holding *count* repeated notes with non-power-of-two *duration*:

```
>>> notetools.make_repeated_notes(3, (1, 12))
Selection(Tuplet(2/3, [c'8, c'8, c'8]),)
```

Set pitch of all notes created to middle C.

Returns list of zero or more newly constructed notes or list of one newly constructed tuplet.

### 18.2.7 `notetools.make_repeated_notes_from_time_signature`

`notetools.make_repeated_notes_from_time_signature` (*time\_signature*, *pitch*="c")

Make repeated notes from *time\_signature*:

```
>>> notetools.make_repeated_notes_from_time_signature((5, 32))
Selection(Note("c'32"), Note("c'32"), Note("c'32"), Note("c'32"), Note("c'32"))
```

Make repeated notes with *pitch* from *time\_signature*:

```
>>> notetools.make_repeated_notes_from_time_signature((5, 32), pitch="d'")
Selection(Note("d'32"), Note("d'32"), Note("d'32"), Note("d'32"), Note("d'32"))
```

Returns list of notes.

### 18.2.8 `notetools.make_repeated_notes_from_time_signatures`

`notetools.make_repeated_notes_from_time_signatures` (*time\_signatures*, *pitch*="c")

Make repeated notes from *time\_signatures*:

```
notetools.make_repeated_notes_from_time_signatures([(2, 8), (3, 32)])
[Selection(Note("c'8"), Note("c'8")), Selection(Note("c'32"), Note("c'32"), Note("c'32"))]
```

Make repeated notes with *pitch* from *time\_signatures*:

```
>>> notetools.make_repeated_notes_from_time_signatures([(2, 8), (3, 32)], pitch="d'")
[Selection(Note("d'8"), Note("d'8")), Selection(Note("d'32"), Note("d'32"), Note("d'32"))]
```

Returns two-dimensional list of note lists.

Use `sequencetools.flatten_sequence()` to flatten output if required.

### 18.2.9 `notetools.make_repeated_notes_with_shorter_notes_at_end`

`notetools.make_repeated_notes_with_shorter_notes_at_end` (*pitch*, *written\_duration*, *total\_duration*, *prolation*=1)

Make repeated notes with *pitch* and *written\_duration* summing to *total\_duration* under *prolation*:

```
>>> args = [0, Duration(1, 16), Duration(1, 4)]
>>> notes = notetools.make_repeated_notes_with_shorter_notes_at_end(*args)
>>> voice = Voice(notes)
```

Fill power-of-two remaining duration with power-of-two notes of lesser written duration:

```
>>> args = [0, Duration(1, 16), Duration(9, 32)]
>>> notes = notetools.make_repeated_notes_with_shorter_notes_at_end(*args)
>>> voice = Voice(notes)
```

Fill non-power-of-two remaining duration with ad hoc tuplet:

```
>>> args = [0, Duration(1, 16), Duration(4, 10)]
>>> notes = notetools.make_repeated_notes_with_shorter_notes_at_end(*args)
>>> voice = Voice(notes)
```

Set *prolation* when making notes in a measure with a non-power-of-two denominator.

Returns list of components.

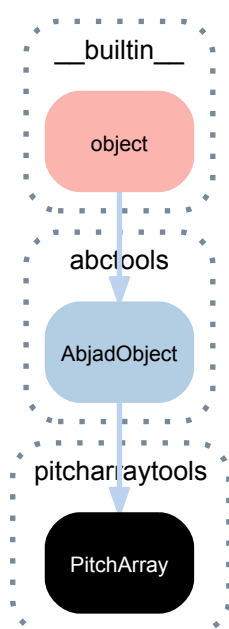




# PITCHARRAYTOOLS

## 19.1 Concrete classes

### 19.1.1 `pitcharraytools.PitchArray`



`class pitcharraytools.PitchArray(*args)`  
Two-dimensional array of pitches.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`PitchArray.cell_tokens_by_row`

`PitchArray.cell_widths_by_row`

`PitchArray.cells`

`PitchArray.columns`

`PitchArray.depth`

PitchArray.**dimensions**  
PitchArray.**has\_voice\_crossing**  
PitchArray.**is\_rectangular**  
PitchArray.**pitches**  
PitchArray.**pitches\_by\_row**  
PitchArray.**rows**  
PitchArray.  
PitchArray.**voice\_crossing\_count**  
PitchArray.**weight**  
PitchArray.**width**

## Methods

PitchArray.**append\_column** (*column*)  
PitchArray.**append\_row** (*row*)  
PitchArray.**apply\_pitches\_by\_row** (*pitch\_lists*)  
PitchArray.**copy\_subarray** (*upper\_left\_pair*, *lower\_right\_pair*)  
PitchArray.**has\_spanning\_cell\_over\_index** (*index*)  
PitchArray.**list\_nonspanning\_subarrays** ()  
    List nonspanning subarrays of pitch array:

```
>>> array = pitcharraytools.PitchArray([
...     [2, 2, 3, 1],
...     [1, 2, 1, 1, 2, 1],
...     [1, 1, 1, 1, 1, 1, 1, 1]])
>>> print array
[ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

```
>>> subarrays = array.list_nonspanning_subarrays()
>>> len(subarrays)
3
```

```
>>> print subarrays[0]
[ ] [ ]
[ ] [ ] [ ]
[ ] [ ] [ ] [ ]
```

```
>>> print subarrays[1]
[ ]
[ ] [ ]
[ ] [ ] [ ]
```

```
>>> print subarrays[2]
[ ]
[ ]
[ ]
```

Returns list.

PitchArray.**pad\_to\_depth** (*depth*)  
PitchArray.**pad\_to\_width** (*width*)  
PitchArray.**pop\_column** (*column\_index*)  
PitchArray.**pop\_row** (*row\_index=-1*)

PitchArray.remove\_row(*row*)

PitchArray.to\_measures(*cell\_duration\_denominator*=8)

Change pitch array to measures with time signatures with numerators equal to row width and denominators equal to *cell\_duration\_denominator* for each row in pitch array:

```
>>> array = pitcharraytools.PitchArray([
...     [1, (2, 1), (-2, -1.5), 2]],
...     [(7, 2), (6, 1), 1])
```

```
>>> print array
[ ] [d'] [bf bqf  ]
[g'   ] [fs'   ] [ ]
```

```
>>> measures = array.to_measures()
```

```
>>> for measure in measures:
...     f(measure)
...
{
    \time 4/8
    r8
    d'8
    <bf bqf>4
}
{
    \time 4/8
    g'4
    fs'8
    r8
}
```

Returns list of measures.

## Static methods

PitchArray.from\_score(*score*, *populate*=True)

Make pitch array from *score*.

**Example 1.** Make empty pitch array from score:

```
>>> score = Score([])
>>> score.append(Staff("c'8 d'8 e'8 f'8"))
>>> score.append(Staff("c'4 d'4"))
>>> score.append(
...     Staff(
...         tuplettools.FixedDurationTuplet(
...             Duration(2, 8), "c'8 d'8 e'8" * 2))
```

```
>>> show(score)
```



```
>>> array = pitcharraytools.PitchArray.from_score(
...     score, populate=False)
```

```
>>> print array
[ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ]
```

**Example 2.** Make populated pitch array from *score*:

```
>>> score = Score([])
>>> score.append(Staff("c'8 d'8 e'8 f'8"))
>>> score.append(Staff("c'4 d'4"))
>>> score.append(
...     Staff(
...         tuplettools.FixedDurationTuplet(
...             Duration(2, 8), "c'8 d'8 e'8") * 2))
```

```
>>> show(score)
```



```
>>> array = pitcharraytools.PitchArray.from_score(
...     score, populate=True)
```

```
>>> print array
[c'      ] [d'      ] [e'      ] [f'      ]
[c'      ] [d'      ] [e'      ] [f'      ]
[c'] [d'      ] [e'] [c'] [d'      ] [e']
```

Returns pitch array.

## Special methods

PitchArray.\_\_add\_\_(arg)

PitchArray.\_\_contains\_\_(arg)

PitchArray.\_\_copy\_\_()

PitchArray.\_\_eq\_\_(arg)

PitchArray.\_\_getitem\_\_(arg)

PitchArray.\_\_iadd\_\_(arg)

Add *arg* in place to self:

```
>>> array_1 = pitcharraytools.PitchArray([[1, 2, 1], [2, 1, 1]])
>>> print array_1
[ ] [ ] [ ]
[ ] [ ] [ ]
```

```
>>> array_2 = pitcharraytools.PitchArray([[3, 4], [4, 3]])
>>> print array_2
[ ] [ ]
[ ] [ ]
```

```
>>> array_3 = pitcharraytools.PitchArray([[1, 1], [1, 1]])
>>> print array_3
[ ] [ ]
[ ] [ ]
```

```
>>> array_1 += array_2
>>> print array_1
[ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ]
```

```
>>> array_1 += array_3
>>> print array_1
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

Returns self.

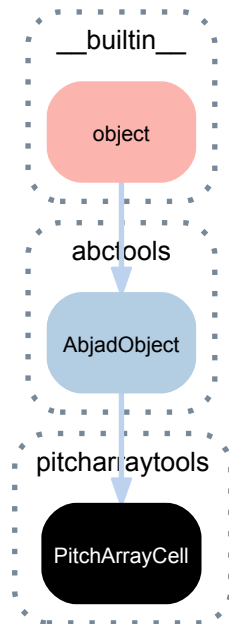
PitchArray.\_\_ne\_\_(arg)

PitchArray.\_\_repr\_\_()

PitchArray.\_\_setitem\_\_(i, arg)

PitchArray.\_\_str\_\_()

### 19.1.2 pitcharraytools.PitchArrayCell



**class** pitcharraytools.**PitchArrayCell** (*cell\_token=None*)

One cell in a pitch array.

```
>>> array = pitcharraytools.PitchArray([[1, 2, 1], [2, 1, 1]])
>>> print array
[ ] [ ] [ ]
[ ] [ ] [ ]
>>> cell = array[0][1]
>>> cell
PitchArrayCell(x2)
```

```
>>> cell.column_indices
(1, 2)
```

```
>>> cell.indices
(0, (1, 2))
```

```
>>> cell.is_first_in_row
False
```

```
>>> cell.is_last_in_row
False
```

```
>>> cell.next
PitchArrayCell(x1)
```

```
>>> cell.parent_array
PitchArray(PitchArrayRow(x1, x2, x1), PitchArrayRow(x2, x1, x1))
```

```
>>> cell.parent_column
PitchArrayColumn(x2, x2)
```

```
>>> cell.parent_row
PitchArrayRow(x1, x2, x1)
```

```
>>> cell.pitches
[]
```

```
>>> cell.prev
PitchArrayCell(x1)
```

```
>>> cell.row_index
0
```

```
>>> cell.token
2
```

```
>>> cell.width
2
```

Returns pitch array cell.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`PitchArrayCell.column_indices`  
Tuple of one or more nonnegative integer indices.

`PitchArrayCell.indices`

`PitchArrayCell.is_first_in_row`

`PitchArrayCell.is_last_in_row`

`PitchArrayCell.next`

`PitchArrayCell.parent_array`

`PitchArrayCell.parent_column`

`PitchArrayCell.parent_row`

`PitchArrayCell.prev`

`PitchArrayCell.row_index`

`PitchArrayCell.token`

`PitchArrayCell.weight`

`PitchArrayCell.width`

## Read/write properties

`PitchArrayCell.pitches`

## Methods

`PitchArrayCell.matches_cell` (*arg*)

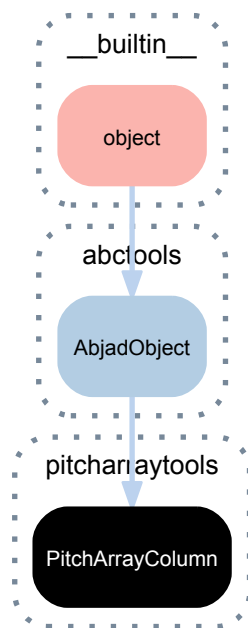
## Special methods

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`PitchArrayCell.__repr__()`  
`PitchArrayCell.__str__()`

### 19.1.3 pitcharraytools.PitchArrayColumn



**class** `pitcharraytools.PitchArrayColumn` (*cells*)  
 Column in a pitch array:

```
>>> array = pitcharraytools.PitchArray([
...     [1, (2, 1), (-1.5, 2)],
...     [(7, 2), (6, 1), 1]])
```

```
>>> print array
[ ] [d'] [bqf  ]
[g'    ] [fs'] [ ]
```

```
>>> array.columns[0]
PitchArrayColumn(x1, g' x2)
```

```
>>> print array.columns[0]
[ ]
[g'    ]
```

Returns pitch array column.

## Bases

- `abctools.AbjadObject`

- `__builtin__.object`

### **Read-only properties**

`PitchArrayColumn.cell_tokens`  
`PitchArrayColumn.cell_widths`  
`PitchArrayColumn.cells`  
`PitchArrayColumn.column_index`  
`PitchArrayColumn.depth`  
`PitchArrayColumn.dimensions`  
`PitchArrayColumn.has_voice_crossing`  
`PitchArrayColumn.is_defective`  
`PitchArrayColumn.parent_array`  
`PitchArrayColumn.pitches`  
`PitchArrayColumn.start_cells`  
`PitchArrayColumn.start_pitches`  
`PitchArrayColumn.stop_cells`  
`PitchArrayColumn.stop_pitches`  
`PitchArrayColumn.weight`  
`PitchArrayColumn.width`

### **Methods**

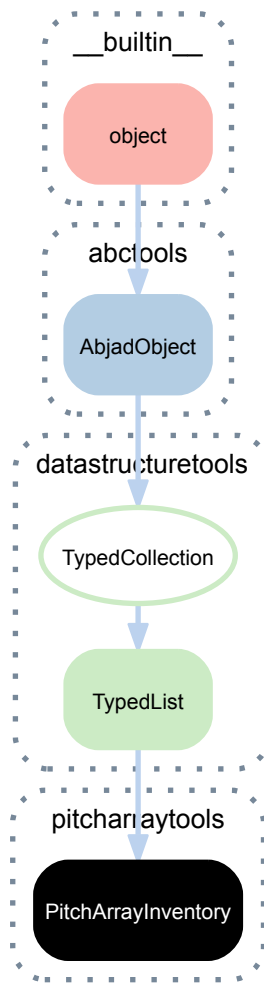
`PitchArrayColumn.append(cell)`  
`PitchArrayColumn.extend(cells)`  
`PitchArrayColumn.remove_pitches()`

### **Special methods**

`PitchArrayColumn.__eq__(arg)`  
`PitchArrayColumn.__getitem__(arg)`  
`PitchArrayColumn.__ne__(arg)`  
`PitchArrayColumn.__repr__()`  
`PitchArrayColumn.__str__()`



### 19.1.4 pitcharraytools.PitchArrayInventory



**class** `pitcharraytools.PitchArrayInventory` (*tokens=None, item\_class=None, name=None*)  
 Ordered collection of pitch arrays:

```

>>> array_1 = pitcharraytools.PitchArray([
...     [1, (2, 1), ([-2, -1.5], 2)],
...     [(7, 2), (6, 1), 1]])

>>> array_2 = pitcharraytools.PitchArray([
...     [1, 1, 1],
...     [1, 1, 1]])

>>> arrays = [array_1, array_2]
>>> inventory = pitcharraytools.PitchArrayInventory(arrays)

>>> print inventory.storage_format
pitcharraytools.PitchArrayInventory([
    pitcharraytools.PitchArray(),
    pitcharraytools.PitchArray()
])
  
```

#### Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`

- `__builtin__.object`

## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedList).append(token)`  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

`(TypedList).count(token)`  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)
3
```

Returns count.

`(TypedList).extend(tokens)`  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

`(TypedList).index(token)`  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('cqi', 'as', 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
```

```
>>> pitch_collection.index("as'")
1
```

Returns index.

(TypedList) **.insert** (*i*, *token*)  
Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['1', 2, 4.3])
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None*, *item\_class=None*, *name=None*)

(TypedList) **.pop** (*i=-1*)  
Aliases list.pop().

(TypedList) **.remove** (*token*)  
Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['0', 1.0, 2, 3.14159])
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()  
Aliases list.reverse().

(TypedList) **.sort** (*cmp=None*, *key=None*, *reverse=False*)  
Aliases list.sort().

PitchArrayInventory **.to\_score** ()  
Make score from pitch arrays in inventory:

```
>>> array_1 = pitcharraytools.PitchArray([
...     [1, (2, 1), (-2, -1.5), 2]],
...     [(7, 2), (6, 1), 1]))
```

```
>>> array_2 = pitcharraytools.PitchArray([
...     [1, 1, 1],
...     [1, 1, 1]])
```

```
>>> arrays = [array_1, array_2]
>>> inventory = pitcharraytools.PitchArrayInventory(arrays)
```

```
>>> score = inventory.to_score()
```

```
>>> show(score)
```



Create one staff per pitch-array row.

Returns score.

## Special methods

(TypedCollection).**\_\_contains\_\_**(*token*)

(TypedList).**\_\_delitem\_\_**(*i*)

Aliases list.**\_\_delitem\_\_**().

(TypedCollection).**\_\_eq\_\_**(*expr*)

(TypedList).**\_\_getitem\_\_**(*i*)

Aliases list.**\_\_getitem\_\_**().

(TypedList).**\_\_iadd\_\_**(*expr*)

Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(  
...     item_class=contexttools.DynamicMark(  
>>> dynamic_collection.append('ppp')  
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format  
datastructuretools.TypedList([  
    contexttools.DynamicMark(  
        'ppp',  
        target_context=stafftools.Staff  
    ),  
    contexttools.DynamicMark(  
        'p',  
        target_context=stafftools.Staff  
    ),  
    contexttools.DynamicMark(  
        'mp',  
        target_context=stafftools.Staff  
    ),  
    contexttools.DynamicMark(  
        'mf',  
        target_context=stafftools.Staff  
    ),  
    contexttools.DynamicMark(  
        'fff',  
        target_context=stafftools.Staff  
    )  
],  
    item_class=contexttools.DynamicMark  
)
```

Returns collection.

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

(TypedCollection).**\_\_ne\_\_**(*expr*)

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(`TypedList`).`__reversed__`()

Aliases `list.__reversed__()`.

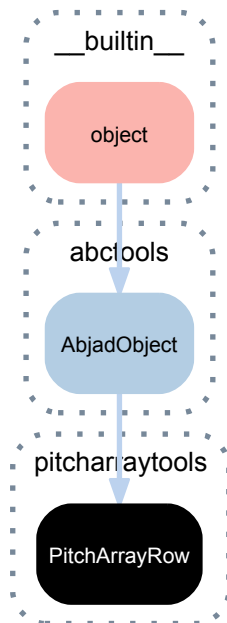
(`TypedList`).`__setitem__`(*i*, *expr*)

Change tokens in *expr* to items and set:

```
>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch('gqs,')
],
    item_class=pitchtools.NamedPitch
)
```

```
>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch("f'"),
    pitchtools.NamedPitch("g'"),
    pitchtools.NamedPitch("a'"),
    pitchtools.NamedPitch("b'"),
    pitchtools.NamedPitch("c'")
],
    item_class=pitchtools.NamedPitch
)
```

### 19.1.5 `pitcharraytools.PitchArrayRow`



`class pitcharraytools.PitchArrayRow`(*cells*)

One row in pitch arra:

```
>>> array = pitcharraytools.PitchArray([[1, 2, 1], [2, 1, 1]])
>>> array[0].cells[0].pitches.append(0)
>>> array[0].cells[1].pitches.append(2)
>>> array[1].cells[2].pitches.append(4)
>>> print array
[c'] [d' ] [ ]
[ ] [ ] [e']
```

```
>>> array[0]
PitchArrayRow(c', d' x2, x1)
```

```
>>> array[0].cell_widths
(1, 2, 1)
```

```
>>> array[0].dimensions
(1, 4)
```

```
>>> array[0].pitches
(NamedPitch("c'"), NamedPitch("d'"))
```

Returns pitch array row.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`PitchArrayRow.cell_tokens`

`PitchArrayRow.cell_widths`

`PitchArrayRow.cells`

`PitchArrayRow.depth`

`PitchArrayRow.dimensions`

`PitchArrayRow.is_defective`

`PitchArrayRow.is_in_range`

`PitchArrayRow.parent_array`

`PitchArrayRow.pitches`

`PitchArrayRow.row_index`

`PitchArrayRow.weight`

`PitchArrayRow.width`

## Read/write properties

`PitchArrayRow.pitch_range`

## Methods

`PitchArrayRow.append(cell_token)`

`PitchArrayRow.apply_pitches(pitch_tokens)`

`PitchArrayRow.copy_subrow(start=None, stop=None)`

`PitchArrayRow.empty_pitches()`

`PitchArrayRow.extend(cell_tokens)`

`PitchArrayRow.has_spanning_cell_over_index(i)`

`PitchArrayRow.index(cell)`

`PitchArrayRow.merge(cells)`

`PitchArrayRow.pad_to_width(width)`

`PitchArrayRow.pop(cell_index)`

`PitchArrayRow.remove(cell)`

`PitchArrayRow.to_measure(cell_duration_denominator=8)`

Change pitch array row to measure with time signature numerator equal to pitch array row width and time signature denominator equal to `cell_duration_denominator`:

```
>>> array = pitcharraytools.PitchArray([
...     [1, (2, 1), ([-2, -1.5], 2)],
...     [(7, 2), (6, 1), 1]])
```

```
>>> print array
[ ] [d'] [bf bqf  ]
[g'      ] [fs'    ] [ ]
```

```
>>> measure = array.rows[0].to_measure()
```

Returns measure.

`PitchArrayRow.withdraw()`

## Special methods

`PitchArrayRow.__add__(arg)`

`PitchArrayRow.__copy__()`

`PitchArrayRow.__eq__(arg)`

`PitchArrayRow.__getitem__(arg)`

`PitchArrayRow.__iadd__(arg)`

`PitchArrayRow.__len__()`

`PitchArrayRow.__ne__(arg)`

`PitchArrayRow.__repr__()`

`PitchArrayRow.__str__()`

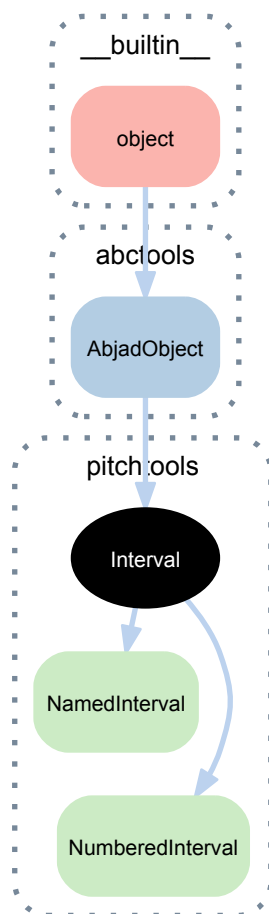




# PITCHTOOLS

## 20.1 Abstract classes

### 20.1.1 `pitchtools.Interval`



**class** `pitchtools.Interval`  
Interval base class.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`Interval.cents`

`Interval.direction_number`

`Interval.direction_string`

`Interval.interval_class`

`Interval.number`

`Interval.semitones`

## Static methods

`Interval.is_named_interval_abbreviation(expr)`

True when *expr* is a named interval abbreviation. Otherwise false:

```
>>> pitchtools.Interval.is_named_interval_abbreviation('+M9')
True
```

The regex `^([+,-]?) (M|m|P|aug|dim) (\d+)$` underlies this predicate.

Returns boolean.

`Interval.is_named_interval_quality_abbreviation(expr)`

True when *expr* is a named-interval quality abbreviation. Otherwise false:

```
>>> pitchtools.Interval.is_named_interval_quality_abbreviation('aug')
True
```

The regex `^M|m|P|aug|dim$` underlies this predicate.

Returns boolean.

## Special methods

`Interval.__abs__()`

`Interval.__eq__(arg)`

`Interval.__float__()`

`Interval.__hash__()`

`Interval.__int__()`

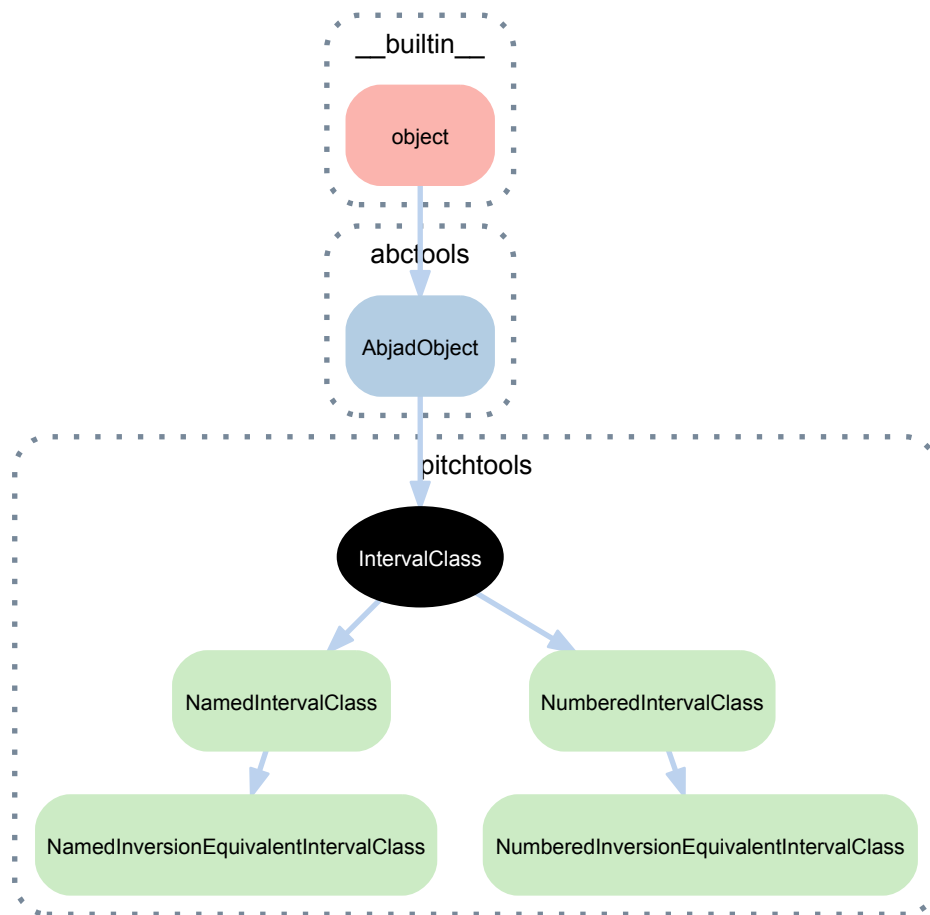
`Interval.__ne__(arg)`

`Interval.__neg__()`

`Interval.__repr__()`

`Interval.__str__()`

### 20.1.2 pitchtools.IntervalClass



**class** `pitchtools.IntervalClass`  
Interval-class base class.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`IntervalClass.number`

#### Special methods

`IntervalClass.__abs__()`

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`IntervalClass.__float__()`

`IntervalClass.__hash__()`

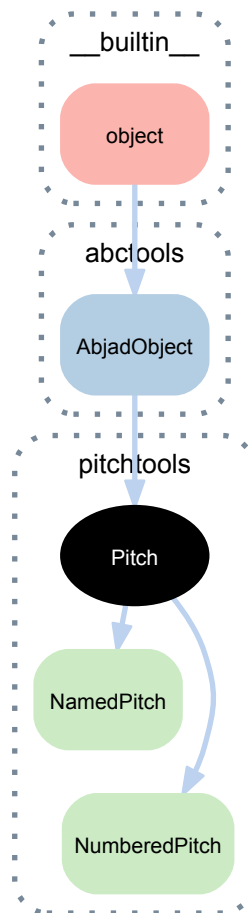
`IntervalClass.__int__()`

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`IntervalClass.__repr__()`

`IntervalClass.__str__()`

### 20.1.3 pitchtools.Pitch



**class** `pitchtools.Pitch`  
 Pitch base class.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`Pitch.accidental`  
 Accidental.

`Pitch.accidental_spelling`  
 Accidental spelling.

```
>>> pitchtools.NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

**Pitch.alteration\_in\_semitones**  
Alteration in semitones.

**Pitch.diatonic\_pitch\_class\_name**  
Diatonic pitch-class name.

**Pitch.diatonic\_pitch\_class\_number**  
Diatonic pitch-class number.

**Pitch.diatonic\_pitch\_name**  
Diatonic pitch name.

**Pitch.diatonic\_pitch\_number**  
Diatonic pitch number.

**Pitch.lilypond\_format**  
LilyPond input format.

**Pitch.named\_pitch**  
Named pitch.

**Pitch.named\_pitch\_class**  
Named pitch-class.

**Pitch.numbered\_pitch**  
Numbered pitch.

**Pitch.numbered\_pitch\_class**  
Numbered pitch-class.

**Pitch.octave**  
Octave indication.

**Pitch.octave\_number**  
Octave number.

**Pitch.pitch\_class\_name**  
Pitch-class name.

**Pitch.pitch\_class\_number**  
Pitch-class number.

**Pitch.pitch\_class\_octave\_label**  
Pitch-class / octave label.

**Pitch.pitch\_name**  
Pitch name.

**Pitch.pitch\_number**  
Pitch number.

## Methods

**Pitch.apply\_accidental** (*accidental=None*)

**Pitch.invert** (*axis=None*)

**Pitch.multiply** (*n=1*)

**Pitch.transpose** (*expr*)

## Static methods

`Pitch.is_diatonic_pitch_name(expr)`

True when *expr* is a diatonic pitch name, otherwise false.

```
>>> pitchtools.Pitch.is_diatonic_pitch_name("c'")
True
```

The regex `^[a-g,A-G])(,+|'+|)'$` underlies this predicate.

Returns boolean.

`Pitch.is_diatonic_pitch_number(expr)`

True when *expr* is a diatonic pitch number, otherwise false.

```
>>> pitchtools.Pitch.is_diatonic_pitch_number(7)
True
```

The diatonic pitch numbers are equal to the set of integers.

Returns boolean.

`Pitch.is_pitch_carrier(expr)`

True when *expr* is an Abjad pitch, note, note-head of chord instance, otherwise false.

```
>>> note = Note("c'4")
>>> pitchtools.Pitch.is_pitch_carrier(note)
True
```

Returns boolean.

`Pitch.is_pitch_class_octave_number_string(expr)`

True when *expr* is a pitch-class / octave number string, otherwise false:

```
>>> pitchtools.Pitch.is_pitch_class_octave_number_string('C#2')
True
```

Quartertone accidentals are supported.

The regex `^([A-G])([#]{1,2}|[b]{1,2}|[#]?[+]|[b]?[~]|)([-]?[0-9]+)$` underlies this predicate.

Returns boolean.

`Pitch.is_pitch_name(expr)`

True *expr* is a pitch name, otherwise false.

```
>>> pitchtools.Pitch.is_pitch_name('c,')
True
```

The regex `^([a-g,A-G])(([s]{1,2}|[f]{1,2}|t?q?[f,s]|)!)?(,+|'+|)'$` underlies this predicate.

Returns boolean.

`Pitch.is_pitch_number(expr)`

True *expr* is a pitch number, otherwise false.

```
>>> pitchtools.Pitch.is_pitch_number(13)
True
```

The pitch numbers are equal to the set of all integers in union with the set of all integers plus of minus 0.5.

Returns boolean.

## Special methods

`Pitch.__abs__()`

```
(AbjadObject).__eq__(expr)
    True when ID of expr equals ID of Abjad object.
    Returns boolean.

Pitch.__float__()

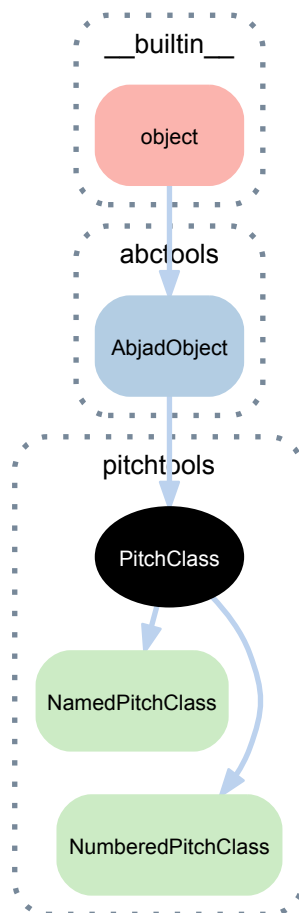
Pitch.__hash__()

Pitch.__int__()

(AbjadObject).__ne__(expr)
    True when ID of expr does not equal ID of Abjad object.
    Returns boolean.

Pitch.__repr__()
```

### 20.1.4 pitchtools.PitchClass



```
class pitchtools.PitchClass
    Pitch-class base class.
```

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`PitchClass.accidental`  
Accidental.

`PitchClass.accidental_spelling`  
Accidental spelling.

```
>>> pitchtools.NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

`PitchClass.alteration_in_semitones`  
Alteration in semitones.

`PitchClass.diatonic_pitch_class_name`  
Diatonic pitch-class name.

`PitchClass.diatonic_pitch_class_number`  
Diatonic pitch-class number.

`PitchClass.named_pitch_class`  
Named pitch-class.

`PitchClass.numbered_pitch_class`  
Numbered pitch-class.

`PitchClass.pitch_class_label`  
Pitch-class label.

`PitchClass.pitch_class_name`  
Pitch-class name.

`PitchClass.pitch_class_number`  
Pitch-class number.

## Methods

`PitchClass.apply_accidental` (*accidental=None*)

`PitchClass.invert` (*axis=None*)

`PitchClass.multiply` (*n=1*)

`PitchClass.transpose` (*expr*)

## Static methods

`PitchClass.is_diatonic_pitch_class_name` (*expr*)  
True when *expr* is a diatonic pitch-class name, otherwise false.

```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_name('c')
True
```

The regex `^[a-g, A-G]$` underlies this predicate.

Returns boolean.

`PitchClass.is_diatonic_pitch_class_number` (*expr*)  
True when *expr* is a diatonic pitch-class number, otherwise false.

```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_number(0)
True
```



```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_number(-5)
False
```

The diatonic pitch-class numbers are equal to the set [0, 1, 2, 3, 4, 5, 6].

Returns boolean.

`PitchClass.is_pitch_class_name(expr)`

True when *expr* is a pitch-class name, otherwise false.

```
>>> pitchtools.PitchClass.is_pitch_class_name('fs')
True
```

The regex `^([a-g,A-G])(( [s]{1,2}|[f]{1,2}|t?q?[fs]|) !?)$` underlies this predicate.

Returns boolean.

`PitchClass.is_pitch_class_number(expr)`

True *expr* is a pitch-class number, otherwise false.

```
>>> pitchtools.PitchClass.is_pitch_class_number(1)
True
```

The pitch-class numbers are equal to the set [0, 0.5, ..., 11, 11.5].

Returns boolean.

## Special methods

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`PitchClass.__hash__()`

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

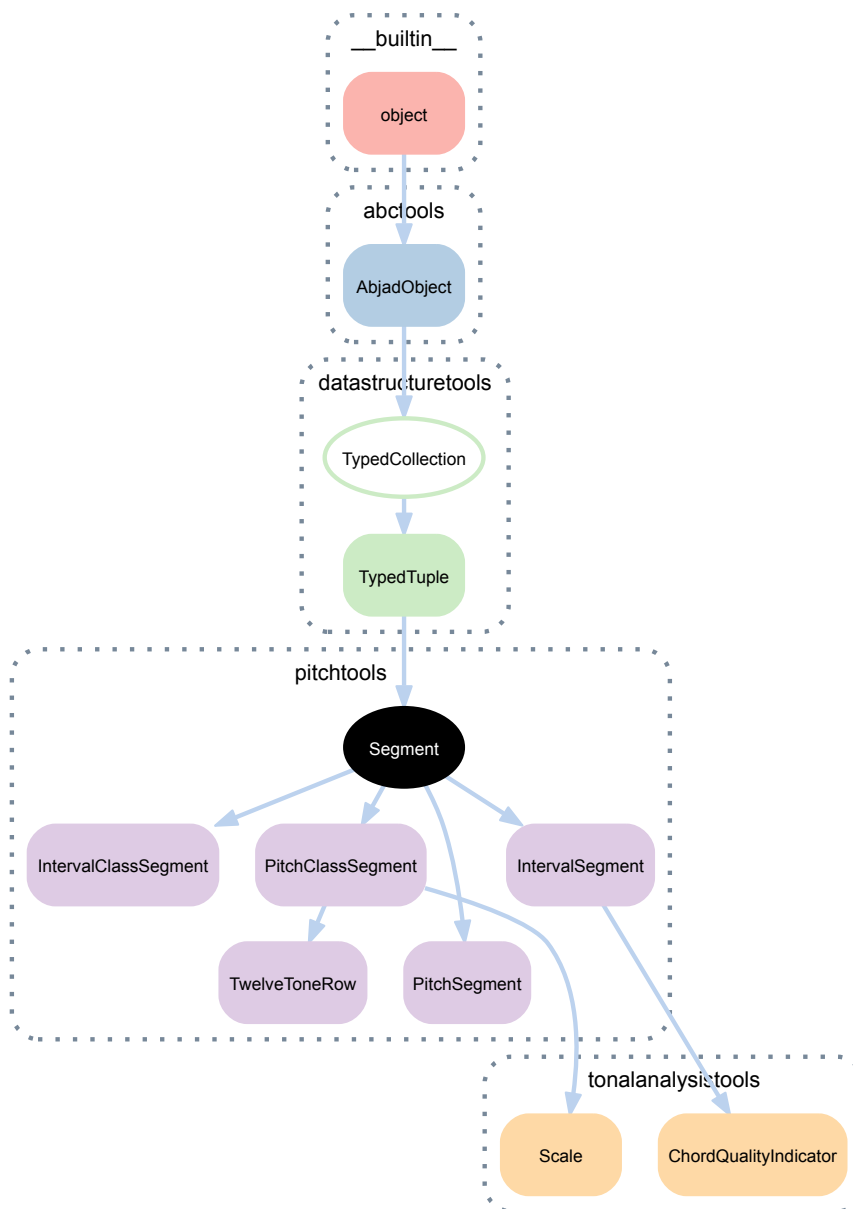
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 20.1.5 pitchtools.Segment



**class** `pitchtools.Segment` (*tokens=None, item\_class=None, name=None*)  
 Music-theoretic segment base class.

### Bases

- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`Segment.has_duplicates`

`(TypedCollection).item_class`

Item class to coerce tokens into.

`(TypedCollection).storage_format`

Storage format of typed tuple.

### Read/write properties

`(TypedCollection).name`

Read / write name of typed tuple.

### Methods

`(TypedTuple).count(token)`

Change *token* to item and return count in collection.

`Segment.from_selection(selection, item_class=None, name=None)`

`(TypedTuple).index(token)`

Change *token* to item and return index in collection.

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

### Special methods

`(TypedTuple).__add__(expr)`

`(TypedTuple).__contains__(token)`

Change *token* to item and return true if item exists in collection.

`(TypedCollection).__eq__(expr)`

`(TypedTuple).__getitem__(i)`

Aliases `tuple.__getitem__()`.

`(TypedTuple).__getslice__(start, stop)`

`(TypedTuple).__hash__()`

`(TypedCollection).__iter__()`

`(TypedCollection).__len__()`

`(TypedTuple).__mul__(expr)`

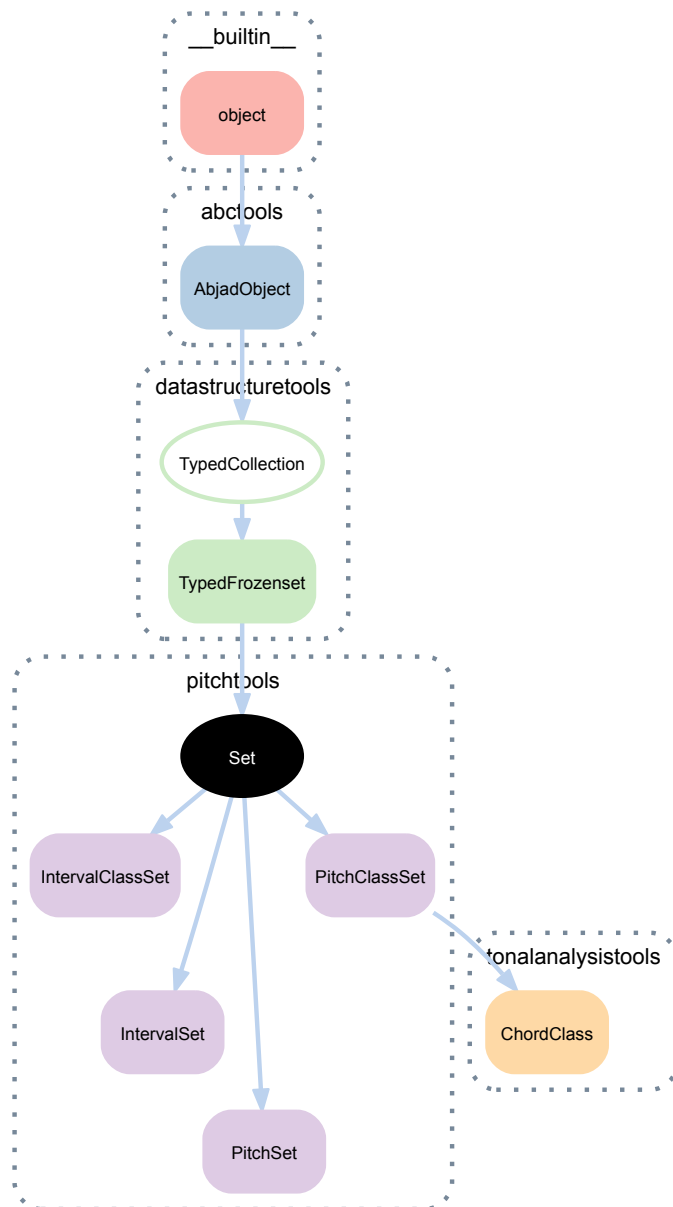
`(TypedCollection).__ne__(expr)`

`Segment.__repr__()`

`(TypedTuple).__rmul__(expr)`

`Segment.__str__()`

## 20.1.6 pitchtools.Set



**class** `pitchtools.Set` (*tokens=None, item\_class=None, name=None*)  
 Music-theoretic set base class.

### Bases

- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(TypedCollection).item_class`  
 Item class to coerce tokens into.

(TypedCollection).**storage\_format**  
Storage format of typed tuple.

### Read/write properties

(TypedCollection).**name**  
Read / write name of typed tuple.

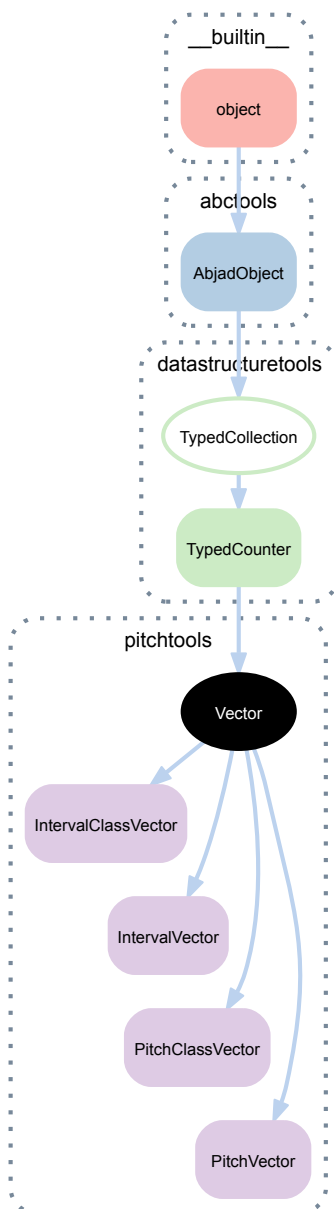
### Methods

(TypedFrozenSet).**copy**()  
(TypedFrozenSet).**difference**(*expr*)  
Set.**from\_selection**(*selection*, *item\_class=None*, *name=None*)  
(TypedFrozenSet).**intersection**(*expr*)  
(TypedFrozenSet).**isdisjoint**(*expr*)  
(TypedFrozenSet).**issubset**(*expr*)  
(TypedFrozenSet).**issuperset**(*expr*)  
(TypedCollection).**new**(*tokens=None*, *item\_class=None*, *name=None*)  
(TypedFrozenSet).**symmetric\_difference**(*expr*)  
(TypedFrozenSet).**union**(*expr*)

### Special methods

(TypedFrozenSet).**\_\_and\_\_**(*expr*)  
(TypedCollection).**\_\_contains\_\_**(*token*)  
(TypedCollection).**\_\_eq\_\_**(*expr*)  
(TypedFrozenSet).**\_\_ge\_\_**(*expr*)  
(TypedFrozenSet).**\_\_gt\_\_**(*expr*)  
(TypedFrozenSet).**\_\_hash\_\_**()  
(TypedCollection).**\_\_iter\_\_**()  
(TypedFrozenSet).**\_\_le\_\_**(*expr*)  
(TypedCollection).**\_\_len\_\_**()  
(TypedFrozenSet).**\_\_lt\_\_**(*expr*)  
(TypedFrozenSet).**\_\_ne\_\_**(*expr*)  
(TypedFrozenSet).**\_\_or\_\_**(*expr*)  
Set.**\_\_repr\_\_**()  
Set.**\_\_str\_\_**()  
(TypedFrozenSet).**\_\_sub\_\_**(*expr*)  
(TypedFrozenSet).**\_\_xor\_\_**(*expr*)

## 20.1.7 pitchtools.Vector



**class** `pitchtools.Vector` (*tokens=None, item\_class=None, name=None*)  
 Music theoretic vector base class.

### Bases

- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(TypedCollection).item_class`  
 Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

### Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

### Methods

`(TypedCounter).clear()`  
`(TypedCounter).copy()`  
`(TypedCounter).elements()`  
`Vector.from_selection(selection, item_class=None, name=None)`  
`(TypedCounter).items()`  
`(TypedCounter).iteritems()`  
`(TypedCounter).iterkeys()`  
`(TypedCounter).intervalvalues()`  
`(TypedCounter).keys()`  
`(TypedCounter).most_common(n=None)`  
`(TypedCollection).new(tokens=None, item_class=None, name=None)`  
`(TypedCounter).subtract(iterable=None, **kwargs)`  
`(TypedCounter).update(iterable=None, **kwargs)`  
`(TypedCounter).values()`  
`(TypedCounter).viewitems()`  
`(TypedCounter).viewkeys()`  
`(TypedCounter).viewvalues()`

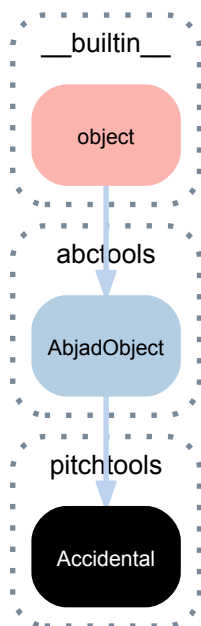
### Special methods

`(TypedCounter).__add__(expr)`  
`(TypedCounter).__and__(expr)`  
`(TypedCollection).__contains__(token)`  
`(TypedCounter).__delitem__(token)`  
`(TypedCollection).__eq__(expr)`  
`(TypedCounter).__getitem__(token)`  
`(TypedCollection).__iter__()`  
`(TypedCollection).__len__()`  
`(TypedCounter).__missing__(token)`  
`(TypedCollection).__ne__(expr)`  
`(TypedCounter).__or__(expr)`  
`Vector.__repr__()`

```
(TypedCounter).__setitem__(token, value)
Vector.__str__()
(TypedCounter).__sub__(expr)
```

## 20.2 Concrete classes

### 20.2.1 pitchtools.Accidental



```
class pitchtools.Accidental (arg='')
    An accidental.
```

```
>>> pitchtools.Accidental('s')
Accidental('s')
```

Accidentals are immutable.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

```
Accidental.alphabetic_accidental_abbreviation
    Alphabetic string:
```

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.alphabetic_accidental_abbreviation
's'
```

Returns string.

```
Accidental.is_adjusted
```

True for all accidentals equal to a nonzero number of semitones. False otherwise:



```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.is_adjusted
True
```

Returns boolean.

#### **Accidental.lilypond\_format**

LilyPond input format of accidental:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.lilypond_format
's'
```

Returns string.

#### **Accidental.name**

Name of accidental:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.name
'sharp'
```

Returns string.

#### **Accidental.semitones**

Semitones of accidental:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.semitones
1
```

Returns number.

#### **Accidental.symbolic\_accidental\_string**

Symbolic string of accidental:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.symbolic_accidental_string
'#'
```

Returns string.

### **Static methods**

#### **Accidental.is\_alphabetic\_accidental\_abbreviation(*expr*)**

True when *expr* is an alphabetic accidental abbreviation. Otherwise false:

```
>>> pitchtools.Accidental.is_alphabetic_accidental_abbreviation('tqs')
True
```

The regex `^([s]{1,2}|[f]{1,2}|t?q?[fs])!?$` underlies this predicate.

Returns boolean.

#### **Accidental.is\_symbolic\_accidental\_string(*expr*)**

True when *expr* is a symbolic accidental string. Otherwise false:

```
>>> pitchtools.Accidental.is_symbolic_accidental_string('#+')
True
```

True on empty string.

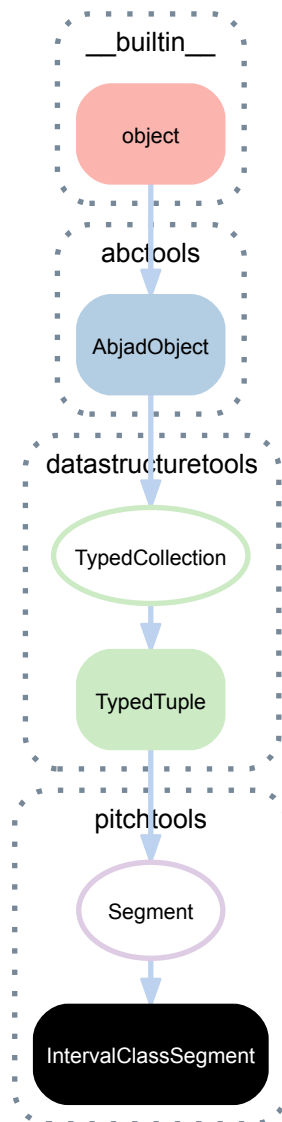
The regex `^([#]{1,2}|[b]{1,2}|[#]?[+]|[b]?[~]|)$` underlies this predicate.

Returns boolean.

### Special methods

`Accidental.__add__(arg)`  
`Accidental.__eq__(arg)`  
`Accidental.__ge__(arg)`  
`Accidental.__gt__(arg)`  
`Accidental.__le__(arg)`  
`Accidental.__lt__(arg)`  
`Accidental.__ne__(arg)`  
`Accidental.__neg__()`  
`Accidental.__nonzero__()`  
`Accidental.__repr__()`  
`Accidental.__str__()`  
`Accidental.__sub__(arg)`

## 20.2.2 pitchtools.IntervalClassSegment



**class** `pitchtools.IntervalClassSegment` (*tokens=None, item\_class=None, name=None*)  
 An interval-class segment.

```
>>> intervals = 'm2 M10 -aug4 P5'
>>> pitchtools.IntervalClassSegment(intervals)
IntervalClassSegment(['+m2', '+M3', '-aug4', '+P5'])
```

Returns interval-class segment.

### Bases

- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`IntervalClassSegment`.**has\_duplicates**

True if segment contains duplicate items:

```
>>> intervals = 'm2 M3 -aug4 m2 P5'
>>> segment = pitchtools.IntervalClassSegment(intervals)
>>> segment.has_duplicates
True
```

```
>>> intervals = 'M3 -aug4 m2 P5'
>>> segment = pitchtools.IntervalClassSegment(intervals)
>>> segment.has_duplicates
False
```

Returns boolean.

`IntervalClassSegment`.**is\_tertian**

True when all diatonic interval-classes in segment are tertian. Otherwise false:

```
>>> interval_class_segment = pitchtools.IntervalClassSegment(
...     tokens=[('major', 3), ('minor', 6), ('major', 6)],
...     item_class=pitchtools.NamedIntervalClass,
...     )
>>> interval_class_segment.is_tertian
True
```

Returns boolean.

(`TypedCollection`).**item\_class**

Item class to coerce tokens into.

(`TypedCollection`).**storage\_format**

Storage format of typed tuple.

## Read/write properties

(`TypedCollection`).**name**

Read / write name of typed tuple.

## Methods

(`TypedTuple`).**count** (*token*)

Change *token* to item and return count in collection.

(`TypedTuple`).**index** (*token*)

Change *token* to item and return index in collection.

(`TypedCollection`).**new** (*tokens=None, item\_class=None, name=None*)

## Class methods

`IntervalClassSegment`.**from\_selection** (*selection, item\_class=None, name=None*)

Initialize interval-class segment from component selection:

```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> pitchtools.IntervalClassSegment.from_selection(selection)
IntervalClassSegment(['-M2', '-M3', '-m3', '+m7', '+M7', '-P5'])
```

Returns interval-class segment.

## Special methods

(TypedTuple) .**\_\_add\_\_** (*expr*)

(TypedTuple) .**\_\_contains\_\_** (*token*)  
Change *token* to item and return true if item exists in collection.

(TypedCollection) .**\_\_eq\_\_** (*expr*)

(TypedTuple) .**\_\_getitem\_\_** (*i*)  
Aliases `tuple.__getitem__()`.

(TypedTuple) .**\_\_getslice\_\_** (*start, stop*)

(TypedTuple) .**\_\_hash\_\_** ()

(TypedCollection) .**\_\_iter\_\_** ()

(TypedCollection) .**\_\_len\_\_** ()

(TypedTuple) .**\_\_mul\_\_** (*expr*)

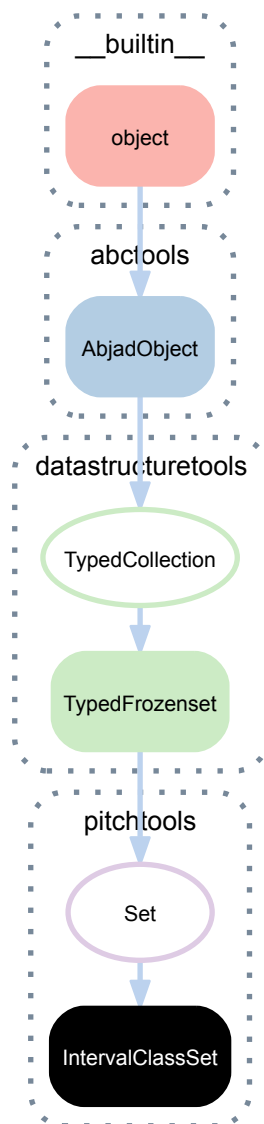
(TypedCollection) .**\_\_ne\_\_** (*expr*)

(Segment) .**\_\_repr\_\_** ()

(TypedTuple) .**\_\_rmul\_\_** (*expr*)

(Segment) .**\_\_str\_\_** ()

## 20.2.3 pitchtools.IntervalClassSet



**class** `pitchtools.IntervalClassSet` (*tokens=None, item\_class=None, name=None*)  
 An interval-class set.

### Bases

- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(TypedCollection).item_class`  
 Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedFrozenSet).copy()`  
`(TypedFrozenSet).difference(expr)`  
`(TypedFrozenSet).intersection(expr)`  
`(TypedFrozenSet).isdisjoint(expr)`  
`(TypedFrozenSet).issubset(expr)`  
`(TypedFrozenSet).issuperset(expr)`  
`(TypedCollection).new(tokens=None, item_class=None, name=None)`  
`(TypedFrozenSet).symmetric_difference(expr)`  
`(TypedFrozenSet).union(expr)`

## Class methods

`IntervalClassSet.from_selection(selection, item_class=None, name=None)`  
Initialize interval set from component selection:

```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> interval_classes = pitchtools.IntervalClassSet.from_selection(
...     selection)
>>> for interval_class in interval_classes:
...     interval_class
...
NamedIntervalClass('-aug4')
NamedIntervalClass('+m7')
NamedIntervalClass('+M3')
NamedIntervalClass('-M2')
NamedIntervalClass('+m3')
NamedIntervalClass('+M7')
NamedIntervalClass('+M6')
NamedIntervalClass('-m3')
NamedIntervalClass('-M3')
NamedIntervalClass('+aug4')
NamedIntervalClass('+P4')
NamedIntervalClass('+M2')
NamedIntervalClass('+P8')
NamedIntervalClass('-P5')
NamedIntervalClass('+P5')
NamedIntervalClass('-M6')
NamedIntervalClass('+m2')
```

Returns interval set.

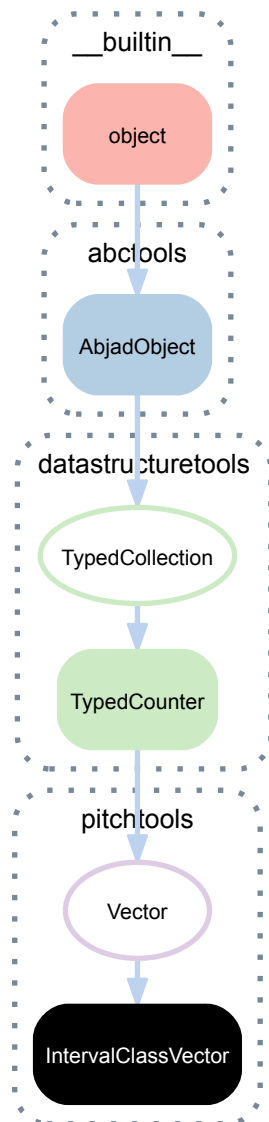
## Special methods

`(TypedFrozenSet).__and__(expr)`  
`(TypedCollection).__contains__(token)`

```
(TypedCollection) .__eq__(expr)
(TypedFrozenSet) .__ge__(expr)
(TypedFrozenSet) .__gt__(expr)
(TypedFrozenSet) .__hash__()
(TypedCollection) .__iter__()
(TypedFrozenSet) .__le__(expr)
(TypedCollection) .__len__()
(TypedFrozenSet) .__lt__(expr)
(TypedFrozenSet) .__ne__(expr)
(TypedFrozenSet) .__or__(expr)
(Set) .__repr__()
(Set) .__str__()
(TypedFrozenSet) .__sub__(expr)
(TypedFrozenSet) .__xor__(expr)
```



## 20.2.4 pitchtools.IntervalClassVector



**class** `pitchtools.IntervalClassVector` (*tokens=None, item\_class=None, name=None*)  
 An interval-class vector.

```

>>> pitch_segment = pitchtools.PitchSegment(
...     tokens=[0, 11, 7, 4, 2, 9, 3, 8, 10, 1, 5, 6],
... )
>>> numbered_interval_class_vector = pitchtools.IntervalClassVector(
...     tokens=pitch_segment,
...     item_class=pitchtools.NumberedInversionEquivalentIntervalClass,
... )
>>> for interval, count in numbered_interval_class_vector.iteritems():
...     print interval, count
...
2 12
3 12
5 12
4 12
6 6
1 12
  
```

Returns interval-class vector.

## Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedCounter).clear()`

`(TypedCounter).copy()`

`(TypedCounter).elements()`

`(TypedCounter).items()`

`(TypedCounter).iteritems()`

`(TypedCounter).iterkeys()`

`(TypedCounter).intervalvalues()`

`(TypedCounter).keys()`

`(TypedCounter).most_common(n=None)`

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

`(TypedCounter).subtract(iterable=None, **kwargs)`

`(TypedCounter).update(iterable=None, **kwargs)`

`(TypedCounter).values()`

`(TypedCounter).viewitems()`

`(TypedCounter).viewkeys()`

`(TypedCounter).viewvalues()`

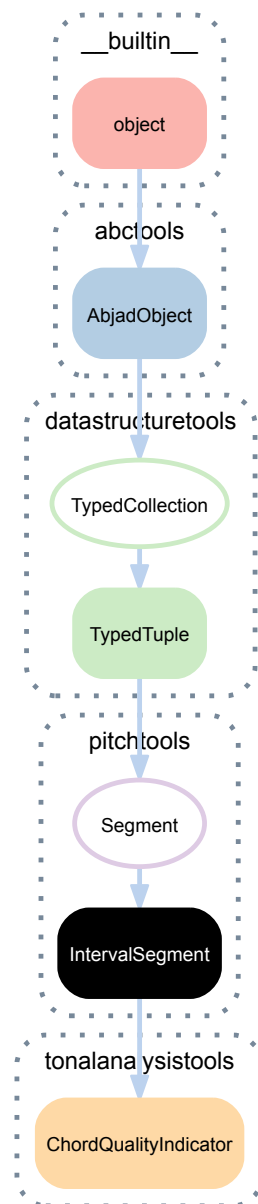
## Class methods

`IntervalClassVector.from_selection(selection, item_class=None, name=None)`

## Special methods

```
(TypedCounter).__add__(expr)
(TypedCounter).__and__(expr)
(TypedCollection).__contains__(token)
(TypedCounter).__delitem__(token)
(TypedCollection).__eq__(expr)
(TypedCounter).__getitem__(token)
(TypedCollection).__iter__()
(TypedCollection).__len__()
(TypedCounter).__missing__(token)
(TypedCollection).__ne__(expr)
(TypedCounter).__or__(expr)
(Vector).__repr__()
(TypedCounter).__setitem__(token, value)
(Vector).__str__()
(TypedCounter).__sub__(expr)
```

## 20.2.5 pitchtools.IntervalSegment



**class** `pitchtools.IntervalSegment` (*tokens=None, item\_class=None, name=None*)  
 An interval segment.

```
>>> intervals = 'm2 M10 -aug4 P5'
>>> pitchtools.IntervalSegment(intervals)
IntervalSegment(['+m2', '+M10', '-aug4', '+P5'])
```

Returns interval segment.

### Bases

- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`IntervalSegment.has_duplicates`

True if segment contains duplicate items:

```
>>> intervals = 'm2 M3 -aug4 m2 P5'
>>> segment = pitchtools.IntervalSegment(intervals)
>>> segment.has_duplicates
True
```

```
>>> intervals = 'M3 -aug4 m2 P5'
>>> segment = pitchtools.IntervalSegment(intervals)
>>> segment.has_duplicates
False
```

Returns boolean.

`(TypedCollection).item_class`

Item class to coerce tokens into.

`IntervalSegment.slope`

The slope of a interval segment is the sum of its intervals divided by its length:

```
>>> pitchtools.IntervalSegment([1, 2]).slope
Multiplier(3, 2)
```

Returns multiplier.

`IntervalSegment.spread`

The maximum interval spanned by any combination of the intervals within a numbered interval segment:

```
>>> pitchtools.IntervalSegment([1, 2, -3, 1, -2, 1]).spread
NumberedInterval(+4.0)
```

```
>>> pitchtools.IntervalSegment([1, 1, 1, 2, -3, -2]).spread
NumberedInterval(+5.0)
```

Returns numbered interval.

`(TypedCollection).storage_format`

Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`

Read / write name of typed tuple.

## Methods

`(TypedTuple).count(token)`

Change *token* to item and return count in collection.

`(TypedTuple).index(token)`

Change *token* to item and return index in collection.

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

`IntervalSegment.rotate(n)`

## Class methods

`IntervalSegment.from_selection(selection, item_class=None, name=None)`

Initialize interval segment from component selection:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> pitchtools.IntervalSegment.from_selection(
...     staff, item_class=pitchtools.NumberedInterval)
IntervalSegment([+2, +2, +1, +2, +2, +2, +1])
```

Returns interval segment.

## Special methods

(TypedTuple).**\_\_add\_\_**(*expr*)

(TypedTuple).**\_\_contains\_\_**(*token*)

Change *token* to item and return true if item exists in collection.

(TypedCollection).**\_\_eq\_\_**(*expr*)

(TypedTuple).**\_\_getitem\_\_**(*i*)

Aliases tuple.**\_\_getitem\_\_**().

(TypedTuple).**\_\_getslice\_\_**(*start*, *stop*)

(TypedTuple).**\_\_hash\_\_**()

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

(TypedTuple).**\_\_mul\_\_**(*expr*)

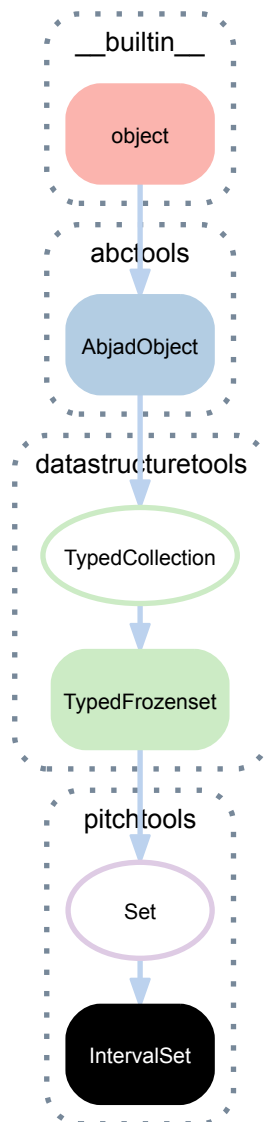
(TypedCollection).**\_\_ne\_\_**(*expr*)

(Segment).**\_\_repr\_\_**()

(TypedTuple).**\_\_rmul\_\_**(*expr*)

(Segment).**\_\_str\_\_**()

## 20.2.6 pitchtools.IntervalSet



**class** `pitchtools.IntervalSet` (*tokens=None, item\_class=None, name=None*)  
 An interval set.

### Bases

- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(TypedCollection).item_class`  
 Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedFrozenSet).copy()`  
`(TypedFrozenSet).difference(expr)`  
`(TypedFrozenSet).intersection(expr)`  
`(TypedFrozenSet).isdisjoint(expr)`  
`(TypedFrozenSet).issubset(expr)`  
`(TypedFrozenSet).issuperset(expr)`  
`(TypedCollection).new(tokens=None, item_class=None, name=None)`  
`(TypedFrozenSet).symmetric_difference(expr)`  
`(TypedFrozenSet).union(expr)`

## Class methods

`IntervalSet.from_selection(selection, item_class=None, name=None)`  
Initialize interval set from component selection:

```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> intervals = pitchtools.IntervalSet.from_selection(
...     selection)
>>> for interval in intervals:
...     interval
...
NamedInterval('+m3')
NamedInterval('+M3')
NamedInterval('+P4')
NamedInterval('+M7')
NamedInterval('-M6')
NamedInterval('+m2')
NamedInterval('+aug11')
NamedInterval('-P5')
NamedInterval('+M13')
NamedInterval('+P8')
NamedInterval('-M3')
NamedInterval('-aug4')
NamedInterval('+M9')
NamedInterval('+m7')
NamedInterval('-M2')
NamedInterval('-m3')
NamedInterval('+P5')
```

Returns interval set.

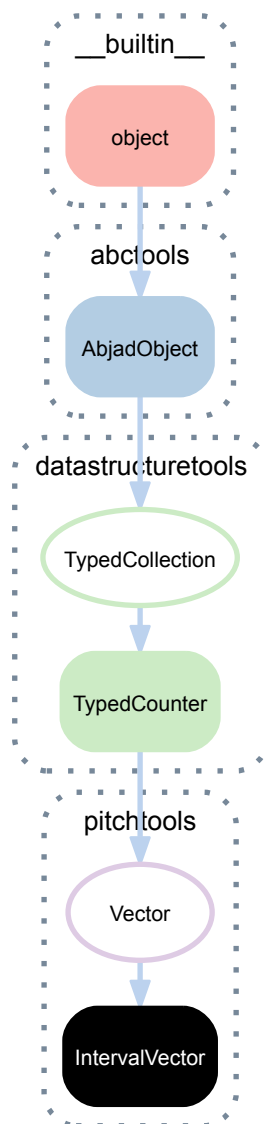
## Special methods

`(TypedFrozenSet).__and__(expr)`  
`(TypedCollection).__contains__(token)`



```
(TypedCollection).__eq__(expr)
(TypedFrozenSet).__ge__(expr)
(TypedFrozenSet).__gt__(expr)
(TypedFrozenSet).__hash__()
(TypedCollection).__iter__()
(TypedFrozenSet).__le__(expr)
(TypedCollection).__len__()
(TypedFrozenSet).__lt__(expr)
(TypedFrozenSet).__ne__(expr)
(TypedFrozenSet).__or__(expr)
(Set).__repr__()
(Set).__str__()
(TypedFrozenSet).__sub__(expr)
(TypedFrozenSet).__xor__(expr)
```

## 20.2.7 pitchtools.IntervalVector



**class** `pitchtools.IntervalVector` (*tokens=None, item\_class=None, name=None*)  
 An interval vector.

```

>>> pitch_segment = pitchtools.PitchSegment(
...     tokens=[0, 11, 7, 4, 2, 9, 3, 8, 10, 1, 5, 6],
... )
>>> numbered_interval_vector = pitchtools.IntervalVector(
...     tokens=pitch_segment,
...     item_class=pitchtools.NumberedInterval,
... )
>>> for interval, count in sorted(numbered_interval_vector.items(),
...     key=lambda x: (x[0].direction_number, x[0].number)):
...     print interval, count
...
-11 1
-10 1
-9 1
-8 2
-7 3
-6 3
-5 4
-4 4
-3 4
-2 5
-1 6
  
```

```
+1 5
+2 5
+3 5
+4 4
+5 3
+6 3
+7 2
+8 2
+9 2
+10 1
```

Returns pitch segment.

## Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedCounter).clear()`

`(TypedCounter).copy()`

`(TypedCounter).elements()`

`(TypedCounter).items()`

`(TypedCounter).iteritems()`

`(TypedCounter).iterkeys()`

`(TypedCounter).intervalues()`

`(TypedCounter).keys()`

`(TypedCounter).most_common(n=None)`

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

`(TypedCounter).subtract(iterable=None, **kwargs)`

`(TypedCounter).update(iterable=None, **kwargs)`

`(TypedCounter).values()`

```
(TypedCounter).viewitems()  
(TypedCounter).viewkeys()  
(TypedCounter).viewvalues()
```

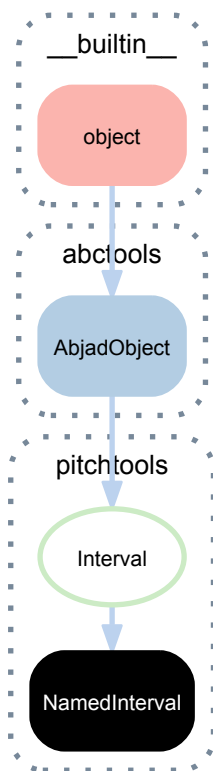
### **Class methods**

```
IntervalVector.from_selection(selection, item_class=None, name=None)
```

### **Special methods**

```
(TypedCounter).__add__(expr)  
(TypedCounter).__and__(expr)  
(TypedCollection).__contains__(token)  
(TypedCounter).__delitem__(token)  
(TypedCollection).__eq__(expr)  
(TypedCounter).__getitem__(token)  
(TypedCollection).__iter__()  
(TypedCollection).__len__()  
(TypedCounter).__missing__(token)  
(TypedCollection).__ne__(expr)  
(TypedCounter).__or__(expr)  
(Vector).__repr__()  
(TypedCounter).__setitem__(token, value)  
(Vector).__str__()  
(TypedCounter).__sub__(expr)
```

## 20.2.8 pitchtools.NamedInterval



**class** `pitchtools.NamedInterval(*args)`  
 A named interval.

```
>>> pitchtools.NamedInterval('+M9')
NamedInterval('+M9')
```

Returns named interval

### Bases

- `pitchtools.Interval`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

```
(Interval).cents
NamedInterval.direction_number
NamedInterval.direction_string
NamedInterval.interval_class
NamedInterval.interval_string
NamedInterval.named_interval_class
NamedInterval.number
NamedInterval.quality_string
NamedInterval.semitones
```

`NamedInterval.staff_spaces`

## Class methods

`NamedInterval.from_pitch_carriers` (*pitch\_carrier\_1*, *pitch\_carrier\_2*)

Calculate named interval from *pitch\_carrier\_1* to *pitch\_carrier\_2*:

```
>>> pitchtools.NamedInterval.from_pitch_carriers(  
...     pitchtools.NamedPitch(-2),  
...     pitchtools.NamedPitch(12),  
... )  
NamedInterval('+M9')
```

Returns named interval.

## Static methods

`(Interval).is_named_interval_abbreviation` (*expr*)

True when *expr* is a named interval abbreviation. Otherwise false:

```
>>> pitchtools.Interval.is_named_interval_abbreviation('+M9')  
True
```

The regex `^([+,-]?)(M|m|P|aug|dim)(\d+)$` underlies this predicate.

Returns boolean.

`(Interval).is_named_interval_quality_abbreviation` (*expr*)

True when *expr* is a named-interval quality abbreviation. Otherwise false:

```
>>> pitchtools.Interval.is_named_interval_quality_abbreviation('aug')  
True
```

The regex `^M|m|P|aug|dim$` underlies this predicate.

Returns boolean.

## Special methods

`NamedInterval.__abs__` ()

`NamedInterval.__add__` (*arg*)

`NamedInterval.__copy__` (\**args*)

`NamedInterval.__eq__` (*arg*)

`NamedInterval.__float__` ()

`NamedInterval.__ge__` (*arg*)

`NamedInterval.__gt__` (*arg*)

`(Interval).__hash__` ()

`NamedInterval.__int__` ()

`NamedInterval.__le__` (*arg*)

`NamedInterval.__lt__` (*arg*)

`NamedInterval.__mul__` (*arg*)

`NamedInterval.__ne__` (*arg*)

`NamedInterval.__neg__` ()

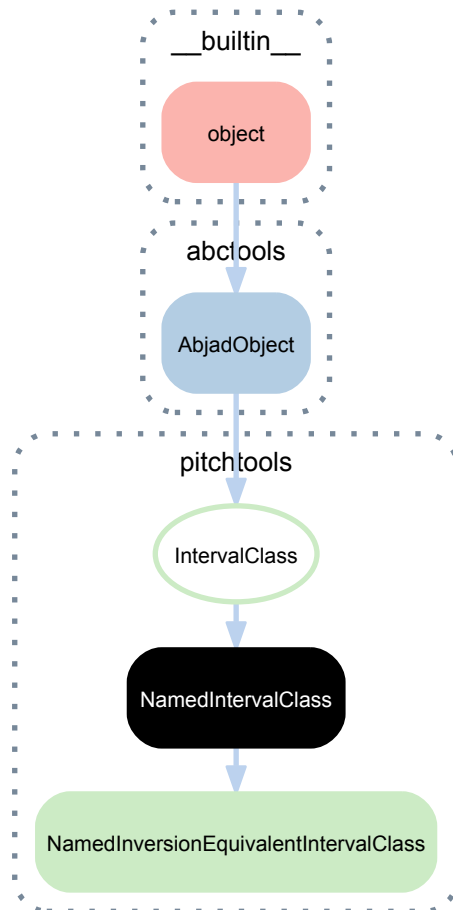
`NamedInterval.__repr__` ()

`NamedInterval.__rmul__(arg)`

`NamedInterval.__str__()`

`NamedInterval.__sub__(arg)`

## 20.2.9 pitchtools.NamedIntervalClass



```
class pitchtools.NamedIntervalClass(*args)
    A named interval-class.
```

```
>>> pitchtools.NamedIntervalClass('-M9')
NamedIntervalClass('-M2')
```

Returns named interval-class.

### Bases

- `pitchtools.IntervalClass`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`NamedIntervalClass.direction_number`

`NamedIntervalClass.direction_symbol`

```
NamedIntervalClass.direction_word  
(IntervalClass).number  
NamedIntervalClass.quality_string
```

### **Class methods**

NamedIntervalClass.**from\_pitch\_carriers** (*pitch\_carrier\_1*, *pitch\_carrier\_2*)  
Calculate named interval-class from *pitch\_carrier\_1* to *pitch\_carrier\_2*:

```
>>> pitchtools.NamedIntervalClass.from_pitch_carriers(  
...     pitchtools.NamedPitch(-2),  
...     pitchtools.NamedPitch(12),  
...     )  
NamedIntervalClass('+M2')
```

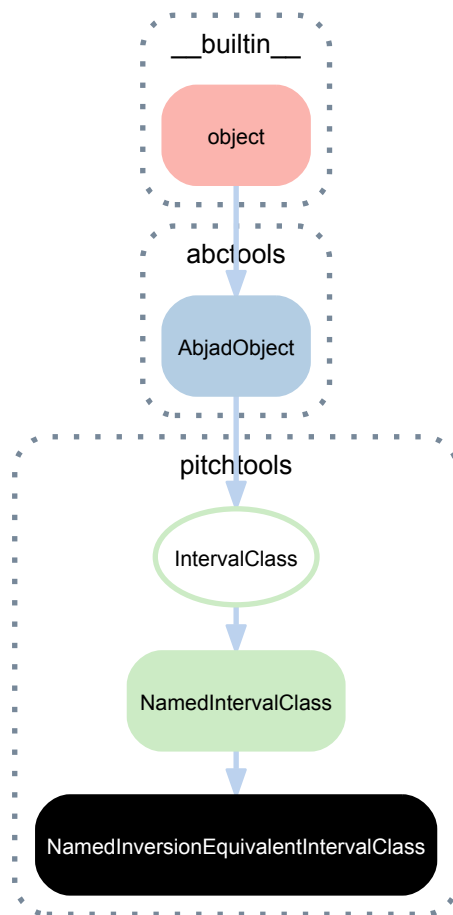
Returns named interval-class.

### **Special methods**

```
NamedIntervalClass.__abs__()  
NamedIntervalClass.__eq__(arg)  
NamedIntervalClass.__float__()  
NamedIntervalClass.__hash__()  
NamedIntervalClass.__int__()  
NamedIntervalClass.__ne__(arg)  
NamedIntervalClass.__repr__()  
NamedIntervalClass.__str__()
```



## 20.2.10 `pitchtools.NamedInversionEquivalentIntervalClass`



**class** `pitchtools.NamedInversionEquivalentIntervalClass` (\*args)  
 An inversion-equivalent diatonic interval-class.

```
>>> pitchtools.NamedInversionEquivalentIntervalClass('-m14')
NamedInversionEquivalentIntervalClass('+M2')
```

Inversion-equivalent diatonic interval-classes are immutable.

### Bases

- `pitchtools.NamedIntervalClass`
- `pitchtools.IntervalClass`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

```
(NamedIntervalClass).direction_number
(NamedIntervalClass).direction_symbol
(NamedIntervalClass).direction_word
(IntervalClass).number
(NamedIntervalClass).quality_string
```

## Class methods

`NamedInversionEquivalentIntervalClass.from_pitch_carriers` (*pitch\_carrier\_1*,  
*pitch\_carrier\_2*)  
Calculate named inversion-equivalent interval-class from *pitch\_carrier\_1* to *pitch\_carrier\_2*:

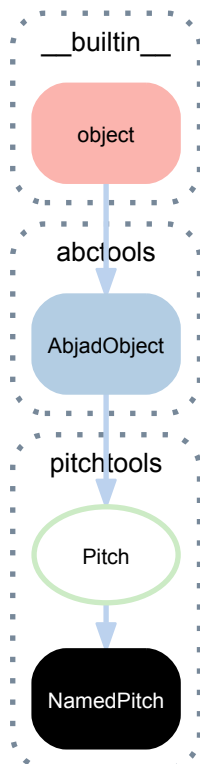
```
>>> pitchtools.NamedInversionEquivalentIntervalClass.from_pitch_carriers(  
...     pitchtools.NamedPitch(-2),  
...     pitchtools.NamedPitch(12),  
... )  
NamedInversionEquivalentIntervalClass('+M2')
```

Returns named inversion-equivalent interval-class.

## Special methods

```
(NamedIntervalClass).__abs__()  
NamedInversionEquivalentIntervalClass.__eq__(arg)  
(NamedIntervalClass).__float__()  
(NamedIntervalClass).__hash__()  
(NamedIntervalClass).__int__()  
NamedInversionEquivalentIntervalClass.__ne__(arg)  
(NamedIntervalClass).__repr__()  
(NamedIntervalClass).__str__()
```

### 20.2.11 pitchtools.NamedPitch



```
class pitchtools.NamedPitch(*args)  
    A named pitch.
```

```
>>> pitchtools.NamedPitch("cs' ' ")
NamedPitch("cs' ' ")
```

Returns named pitch.

## Bases

- `pitchtools.Pitch`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`NamedPitch.accidental`  
Accidental.

```
>>> pitchtools.NamedPitch("cs' ' ").accidental
Accidental('s')
```

Returns accidental.

`(Pitch).accidental_spelling`  
Accidental spelling.

```
>>> pitchtools.NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

`NamedPitch.alteration_in_semitones`  
Alteration in semitones.

```
>>> pitchtools.NamedPitch("cs' ' ").alteration_in_semitones
1
```

Returns integer or float.

`NamedPitch.diatonic_pitch_class_name`  
Diatonic pitch-class name.

```
>>> pitchtools.NamedPitch("cs' ' ").diatonic_pitch_class_name
'c'
```

Returns string.

`NamedPitch.diatonic_pitch_class_number`  
Diatonic pitch-class number.

```
>>> pitchtools.NamedPitch("cs' ' ").diatonic_pitch_class_number
0
```

Returns integer.

`NamedPitch.diatonic_pitch_name`  
Diatonic pitch name.

```
>>> pitchtools.NamedPitch("cs' ' ").diatonic_pitch_name
"cs' ' "
```

Returns string.

`NamedPitch.diatonic_pitch_number`  
Diatonic pitch number.

```
>>> pitchtools.NamedPitch("cs'").diatonic_pitch_number
7
```

Returns integer.

**NamedPitch.lilypond\_format**  
LilyPond input format.

```
>>> pitchtools.NamedPitch("cs'").lilypond_format
"cs' "
```

Returns string.

**NamedPitch.named\_pitch**  
Named pitch.

```
>>> pitchtools.NamedPitch("cs'").named_pitch
NamedPitch("cs' ")
```

Returns named pitch.

**NamedPitch.named\_pitch\_class**  
Named pitch-class.

```
>>> pitchtools.NamedPitch("cs'").named_pitch_class
NamedPitchClass('cs')
```

Returns named pitch-class.

**NamedPitch.numbered\_pitch**  
Numbered pitch.

```
>>> pitchtools.NamedPitch("cs'").numbered_pitch
NumberedPitch(13)
```

Returns numbered pitch.

**NamedPitch.numbered\_pitch\_class**  
Numbered pitch-class.

```
>>> pitchtools.NamedPitch("cs'").numbered_pitch_class
NumberedPitchClass(1)
```

Returns numbered pitch-class.

**NamedPitch.octave**  
Octave indication.

```
>>> pitchtools.NamedPitch("cs'").octave
Octave(5)
```

Returns octave.

**NamedPitch.octave\_number**  
Integer octave number.

```
>>> pitchtools.NamedPitch("cs'").octave_number
5
```

Returns integer.

**NamedPitch.pitch\_class\_name**  
Pitch-class name.

```
>>> pitchtools.NamedPitch("cs'").pitch_class_name
'cs'
```

Returns string.

`NamedPitch.pitch_class_number`  
Pitch-class number.

```
>>> pitchtools.NamedPitch("cs' ").pitch_class_number
1
```

Returns integer or float.

`NamedPitch.pitch_class_octave_label`  
Pitch-class / octave label.

```
>>> pitchtools.NamedPitch("cs' ").pitch_class_octave_label
'C#5'
```

Returns string.

`NamedPitch.pitch_name`  
Pitch name.

```
>>> pitchtools.NamedPitch("cs' ").pitch_name
"cs' "
```

Returns string.

`NamedPitch.pitch_number`  
Pitch-class number.

```
>>> pitchtools.NamedPitch("cs' ").pitch_number
13
```

```
>>> pitchtools.NamedPitch("cff' ").pitch_number
10
```

Returns integer or float.

## Methods

`NamedPitch.apply_accidental (accidental=None)`  
Apply *accidental* to named pitch.

```
>>> pitchtools.NamedPitch("cs' ").apply_accidental('s')
NamedPitch("css' ")
```

Returns new named pitch.

`NamedPitch.invert (axis=None)`  
Invert pitch around *axis*.

Not yet implemented.

`NamedPitch.multiply (n=1)`  
Multiply pitch-class by *n*, maintaining octave.

```
>>> pitchtools.NamedPitch('cs,').multiply(3)
NamedPitch('ef,')
```

Emit new numbered pitch.

`NamedPitch.respell_with_flats()`

`NamedPitch.respell_with_sharps()`

`NamedPitch.transpose (expr)`  
Transpose by *expr*.

Not yet implemented.

## Static methods

`(Pitch).is_diatonic_pitch_name(expr)`  
True when *expr* is a diatonic pitch name, otherwise false.

```
>>> pitchtools.Pitch.is_diatonic_pitch_name("c'")
True
```

The regex `^[a-g,A-G])(,+|'+|)'$` underlies this predicate.

Returns boolean.

`(Pitch).is_diatonic_pitch_number(expr)`  
True when *expr* is a diatonic pitch number, otherwise false.

```
>>> pitchtools.Pitch.is_diatonic_pitch_number(7)
True
```

The diatonic pitch numbers are equal to the set of integers.

Returns boolean.

`(Pitch).is_pitch_carrier(expr)`  
True when *expr* is an Abjad pitch, note, note-head of chord instance, otherwise false.

```
>>> note = Note("c'4")
>>> pitchtools.Pitch.is_pitch_carrier(note)
True
```

Returns boolean.

`(Pitch).is_pitch_class_octave_number_string(expr)`  
True when *expr* is a pitch-class / octave number string, otherwise false:

```
>>> pitchtools.Pitch.is_pitch_class_octave_number_string('C#2')
True
```

Quartertone accidentals are supported.

The regex `^([A-G])([#]{1,2}|[b]{1,2}|[#]?[+]|[b]?[~]|)([-]?[0-9]+)$` underlies this predicate.

Returns boolean.

`(Pitch).is_pitch_name(expr)`  
True *expr* is a pitch name, otherwise false.

```
>>> pitchtools.Pitch.is_pitch_name('c,')
True
```

The regex `^[a-g,A-G])(([s]{1,2}|[f]{1,2}|t?q?[f,s]|)!)?(,+|'+|)'$` underlies this predicate.

Returns boolean.

`(Pitch).is_pitch_number(expr)`  
True *expr* is a pitch number, otherwise false.

```
>>> pitchtools.Pitch.is_pitch_number(13)
True
```

The pitch numbers are equal to the set of all integers in union with the set of all integers plus of minus 0.5.

Returns boolean.

## Special methods

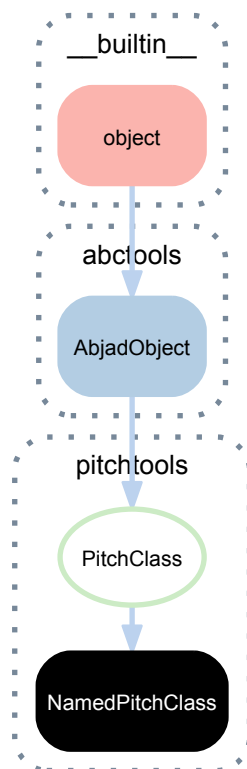
`NamedPitch.__abs__()`

```

NamedPitch.__add__(interval)
NamedPitch.__copy__(*args)
NamedPitch.__eq__(arg)
NamedPitch.__float__()
NamedPitch.__ge__(arg)
NamedPitch.__gt__(arg)
NamedPitch.__hash__()
NamedPitch.__int__()
NamedPitch.__le__(arg)
NamedPitch.__lt__(arg)
NamedPitch.__ne__(arg)
NamedPitch.__repr__()
NamedPitch.__str__()
NamedPitch.__sub__(arg)

```

### 20.2.12 pitchtools.NamedPitchClass



```

class pitchtools.NamedPitchClass (expr)
    A named pitch-class.

```

```

>>> pitchtools.NamedPitchClass('cs')
NamedPitchClass('cs')

```

```

>>> pitchtools.NamedPitchClass(14)
NamedPitchClass('d')

```

```
>>> pitchtools.NamedPitchClass(pitchtools.NamedPitch('g,'))
NamedPitchClass('g')
```

```
>>> pitchtools.NamedPitchClass(pitchtools.NumberedPitch(15))
NamedPitchClass('ef')
```

```
>>> pitchtools.NamedPitchClass(pitchtools.NumberedPitchClass(4))
NamedPitchClass('e')
```

```
>>> pitchtools.NamedPitchClass('C#5')
NamedPitchClass('cs')
```

```
>>> pitchtools.NamedPitchClass(Note("a'8."))
NamedPitchClass('a')
```

Returns named pitch-class.

## Bases

- `pitchtools.PitchClass`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`NamedPitchClass.accidental`  
Accidental.

```
>>> pitchtools.NamedPitchClass('cs').accidental
Accidental('s')
```

Returns accidental.

`(PitchClass).accidental_spelling`  
Accidental spelling.

```
>>> pitchtools.NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

`NamedPitchClass.alteration_in_semitones`  
Alteration in semitones.

```
>>> pitchtools.NamedPitchClass('cs').alteration_in_semitones
1
```

Returns integer or float.

`NamedPitchClass.diatonic_pitch_class_name`  
Diatonic pitch-class name.

```
>>> pitchtools.NamedPitchClass('cs').diatonic_pitch_class_name
'c'
```

Returns string.

`NamedPitchClass.diatonic_pitch_class_number`  
Diatonic pitch-class number.

```
>>> pitchtools.NamedPitchClass('cs').diatonic_pitch_class_number
0
```

Returns integer.



`NamedPitchClass.named_pitch_class`  
 Named pitch-class.

```
>>> pitchtools.NamedPitchClass('cs').named_pitch_class
NamedPitchClass('cs')
```

Returns named pitch-class.

`NamedPitchClass.numbered_pitch_class`  
 Numbered pitch-class.

```
>>> pitchtools.NamedPitchClass('cs').numbered_pitch_class
NumberedPitchClass(1)
```

Returns numbered pitch-class.

`NamedPitchClass.pitch_class_label`  
 Pitch-class label.

```
>>> pitchtools.NamedPitchClass('cs').pitch_class_label
'C#'
```

Returns string.

`NamedPitchClass.pitch_class_name`  
 Pitch-class name.

```
>>> pitchtools.NamedPitchClass('cs').pitch_class_name
'cs'
```

Returns string.

`NamedPitchClass.pitch_class_number`  
 Pitch-class number.

```
>>> pitchtools.NamedPitchClass('cs').pitch_class_number
1
```

Returns integer or float.

## Methods

`NamedPitchClass.apply_accidental (accidental)`  
 Apply *accidental*:

```
>>> pitchtools.NamedPitchClass('cs').apply_accidental('qs')
NamedPitchClass('ctqs')
```

Emit new named pitch-class.

`NamedPitchClass.invert ()`  
 Invert pitch-class.

Not yet implemented.

`NamedPitchClass.multiply (n=1)`  
 Multiply pitch-class number by *n*:

```
>>> pitchtools.NamedPitchClass('cs').multiply(3)
NamedPitchClass('ef')
```

Emit new named pitch-class.

`NamedPitchClass.transpose (n)`  
 Transpose named pitch-class by *named\_interval*:

```
>>> pitchtools.NamedPitchClass('cs').transpose(
...     pitchtools.NamedInterval('major', 2))
NamedPitchClass('ds')
```

Emit new named pitch-class.

## Static methods

`(PitchClass).is_diatonic_pitch_class_name(expr)`  
True when *expr* is a diatonic pitch-class name, otherwise false.

```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_name('c')
True
```

The regex `^[a-g, A-G]$` underlies this predicate.

Returns boolean.

`(PitchClass).is_diatonic_pitch_class_number(expr)`  
True when *expr* is a diatonic pitch-class number, otherwise false.

```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_number(0)
True
```

```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_number(-5)
False
```

The diatonic pitch-class numbers are equal to the set `[0, 1, 2, 3, 4, 5, 6]`.

Returns boolean.

`(PitchClass).is_pitch_class_name(expr)`  
True when *expr* is a pitch-class name, otherwise false.

```
>>> pitchtools.PitchClass.is_pitch_class_name('fs')
True
```

The regex `^([a-g, A-G])((([s]{1,2}|[f]{1,2}|t?q?[fs]|)!)?)$` underlies this predicate.

Returns boolean.

`(PitchClass).is_pitch_class_number(expr)`  
True *expr* is a pitch-class number, otherwise false.

```
>>> pitchtools.PitchClass.is_pitch_class_number(1)
True
```

The pitch-class numbers are equal to the set `[0, 0.5, ..., 11, 11.5]`.

Returns boolean.

## Special methods

`NamedPitchClass.__abs__()`

`NamedPitchClass.__add__(named_interval)`

`NamedPitchClass.__copy__(*args)`

`NamedPitchClass.__eq__(expr)`

`NamedPitchClass.__float__()`

`(PitchClass).__hash__()`

`NamedPitchClass.__int__()`

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

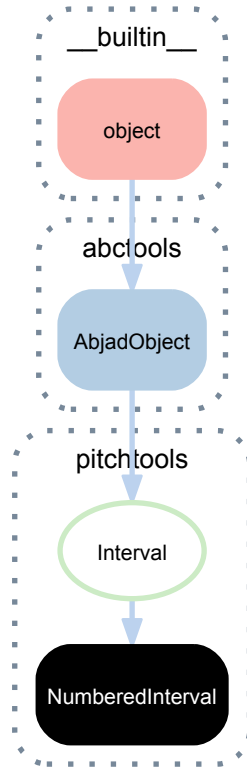
Returns boolean.

`NamedPitchClass.__repr__()`

NamedPitchClass.\_\_str\_\_()

NamedPitchClass.\_\_sub\_\_(arg)

### 20.2.13 pitchtools.NumberedInterval



```
class pitchtools.NumberedInterval(arg)
    A numbered interval.
```

```
>>> numbered_interval = pitchtools.NumberedInterval(-14)
>>> numbered_interval
NumberedInterval(-14)
```

Returns numbered interval.

#### Bases

- `pitchtools.Interval`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(Interval).cents`

`NumberedInterval.direction_number`

Numeric sign:

```
>>> pitchtools.NumberedInterval(-14).direction_number
-1
```

Returns integer.

`(Interval).direction_string`

`(Interval).interval_class`

`NumberedInterval.number`

`NumberedInterval.numbered_interval_number`

Chromatic interval number:

```
>>> pitchtools.NumberedInterval(-14).numbered_interval_number
-14
```

Returns integer or float.

`NumberedInterval.semitones`

## Class methods

`NumberedInterval.from_pitch_carriers(pitch_carrier_1, pitch_carrier_2)`

Calculate numbered interval from *pitch\_carrier\_1* to *pitch\_carrier\_2*:

```
>>> pitchtools.NumberedInterval.from_pitch_carriers(
...     pitchtools.NamedPitch(-2),
...     pitchtools.NamedPitch(12),
... )
NumberedInterval(+14)
```

Returns numbered interval.

## Static methods

`(Interval).is_named_interval_abbreviation(expr)`

True when *expr* is a named interval abbreviation. Otherwise false:

```
>>> pitchtools.Interval.is_named_interval_abbreviation('+M9')
True
```

The regex `^([+,-]?)(M|m|P|aug|dim)(\d+)$` underlies this predicate.

Returns boolean.

`(Interval).is_named_interval_quality_abbreviation(expr)`

True when *expr* is a named-interval quality abbreviation. Otherwise false:

```
>>> pitchtools.Interval.is_named_interval_quality_abbreviation('aug')
True
```

The regex `^M|m|P|aug|dim$` underlies this predicate.

Returns boolean.

## Special methods

`NumberedInterval.__abs__()`

`NumberedInterval.__add__(arg)`

`NumberedInterval.__copy__()`

`NumberedInterval.__eq__(arg)`

`NumberedInterval.__float__()`

`NumberedInterval.__ge__(arg)`

`NumberedInterval.__gt__(arg)`

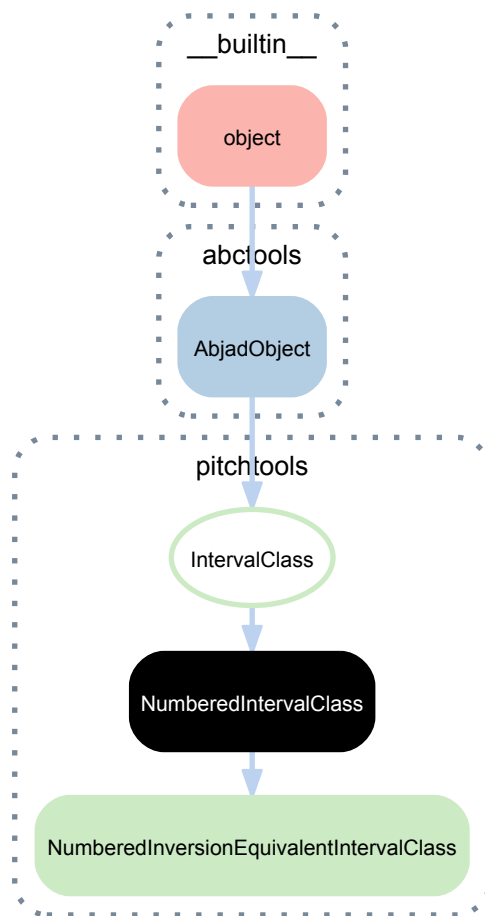
`NumberedInterval.__hash__()`

```

NumberedInterval.__int__()
NumberedInterval.__le__(arg)
NumberedInterval.__lt__(arg)
NumberedInterval.__ne__(arg)
NumberedInterval.__neg__()
NumberedInterval.__repr__()
NumberedInterval.__str__()
NumberedInterval.__sub__(arg)

```

## 20.2.14 pitchtools.NumberedIntervalClass



**class** pitchtools.**NumberedIntervalClass** (*token*)  
A numbered interval-class.

```

>>> pitchtools.NumberedIntervalClass(-14)
NumberedIntervalClass(-2)

```

Returns numbered interval-class.

### Bases

- pitchtools.IntervalClass
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

`NumberedIntervalClass.direction_number`  
`NumberedIntervalClass.direction_symbol`  
`NumberedIntervalClass.direction_word`  
`(IntervalClass).number`

## Class methods

`NumberedIntervalClass.from_pitch_carriers` (*pitch\_carrier\_1*, *pitch\_carrier\_2*)  
Calculate numbered interval-class from *pitch\_carrier\_1* to *pitch\_carrier\_2*:

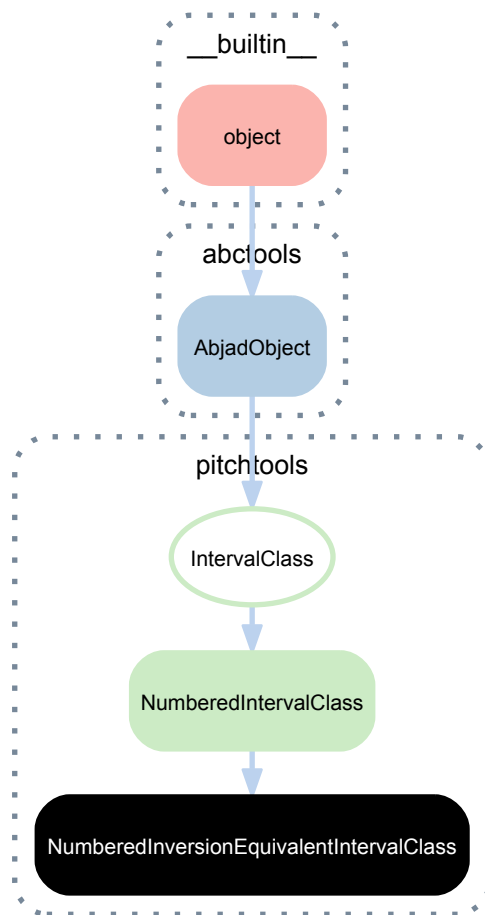
```
>>> pitchtools.NumberedIntervalClass.from_pitch_carriers(  
...     pitchtools.NamedPitch(-2),  
...     pitchtools.NamedPitch(12),  
... )  
NumberedIntervalClass(+2)
```

Returns numbered interval-class.

## Special methods

`NumberedIntervalClass.__abs__()`  
`NumberedIntervalClass.__eq__(arg)`  
`NumberedIntervalClass.__float__()`  
`(IntervalClass).__hash__()`  
`NumberedIntervalClass.__int__()`  
`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.  
`(IntervalClass).__repr__()`  
`(IntervalClass).__str__()`

## 20.2.15 pitchtools.NumberedInversionEquivalentIntervalClass



**class** `pitchtools.NumberedInversionEquivalentIntervalClass` (*interval\_class\_token*)  
 A numbered inversion-equivalent interval-class.

```
>>> pitchtools.NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(1)
```

Returns numbered inversion-equivalent interval-class.

### Bases

- `pitchtools.NumberedIntervalClass`
- `pitchtools.IntervalClass`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

```
(NumberedIntervalClass).direction_number
(NumberedIntervalClass).direction_symbol
(NumberedIntervalClass).direction_word
(IntervalClass).number
```

## Class methods

(NumberedIntervalClass).**from\_pitch\_carriers**(*pitch\_carrier\_1*, *pitch\_carrier\_2*)  
Calculate numbered interval-class from *pitch\_carrier\_1* to *pitch\_carrier\_2*:

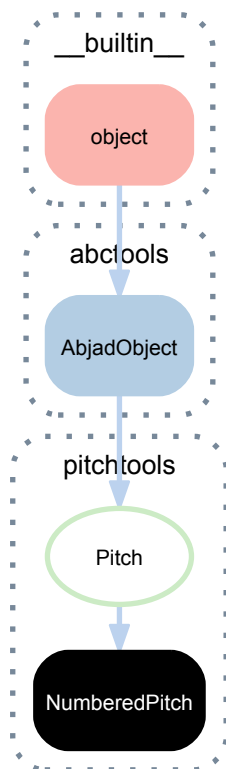
```
>>> pitchtools.NumberedIntervalClass.from_pitch_carriers(  
...     pitchtools.NamedPitch(-2),  
...     pitchtools.NamedPitch(12),  
... )  
NumberedIntervalClass(+2)
```

Returns numbered interval-class.

## Special methods

NumberedInversionEquivalentIntervalClass.**\_\_abs\_\_**()  
NumberedInversionEquivalentIntervalClass.**\_\_copy\_\_**()  
NumberedInversionEquivalentIntervalClass.**\_\_eq\_\_**(*arg*)  
(NumberedIntervalClass).**\_\_float\_\_**()  
NumberedInversionEquivalentIntervalClass.**\_\_hash\_\_**()  
(NumberedIntervalClass).**\_\_int\_\_**()  
NumberedInversionEquivalentIntervalClass.**\_\_ne\_\_**(*arg*)  
NumberedInversionEquivalentIntervalClass.**\_\_neg\_\_**()  
NumberedInversionEquivalentIntervalClass.**\_\_repr\_\_**()  
NumberedInversionEquivalentIntervalClass.**\_\_str\_\_**()

### 20.2.16 pitchtools.NumberedPitch





**class** `pitchtools.NumberedPitch` (*expr*)  
 A numbered pitch.

```
>>> pitchtools.NumberedPitch(13)
NumberedPitch(13)
```

Returns numbered pitch.

## Bases

- `pitchtools.Pitch`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`NumberedPitch.accidental`  
 Accidental.

```
>>> pitchtools.NumberedPitchClass(13).accidental
Accidental('s')
```

Returns accidental.

`(Pitch).accidental_spelling`  
 Accidental spelling.

```
>>> pitchtools.NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

`NumberedPitch.alteration_in_semitones`  
 Alteration in semitones.

```
>>> pitchtools.NumberedPitchClass(13).alteration_in_semitones
1
```

Returns integer or float.

`NumberedPitch.diatonic_pitch_class_name`  
 Diatonic pitch-class name.

```
>>> pitchtools.NumberedPitch(13).diatonic_pitch_class_name
'c'
```

Returns string.

`NumberedPitch.diatonic_pitch_class_number`  
 Diatonic pitch-class number.

```
>>> pitchtools.NumberedPitch(13).diatonic_pitch_class_number
0
```

Returns integer.

`NumberedPitch.diatonic_pitch_name`  
 Diatonic pitch name.

```
>>> pitchtools.NumberedPitch(13).diatonic_pitch_name
'c' '' "
```

Returns string.

`NumberedPitch.diatonic_pitch_number`

Diatonic pitch-class number.

```
>>> pitchtools.NumberedPitch(13).diatonic_pitch_number
7
```

Returns integer.

`NumberedPitch.lilypond_format`

LilyPond input format.

```
>>> pitchtools.NumberedPitch(13).lilypond_format
"cs' '"
```

Returns string.

`NumberedPitch.named_pitch`

Named pitch.

```
>>> pitchtools.NumberedPitch(13).named_pitch
NamedPitch("cs' ' ")
```

Returns named pitch.

`NumberedPitch.named_pitch_class`

Named pitch-class.

```
>>> pitchtools.NumberedPitch(13).named_pitch_class
NamedPitchClass('cs')
```

Returns named pitch-class.

`NumberedPitch.numbered_pitch`

Numbered pitch.

```
>>> pitchtools.NumberedPitch(13).numbered_pitch
NumberedPitch(13)
```

Returns numbered pitch.

`NumberedPitch.numbered_pitch_class`

Numbered pitch-class.

```
>>> pitchtools.NumberedPitch(13).numbered_pitch_class
NumberedPitchClass(1)
```

Returns numbered pitch-class.

`NumberedPitch.octave`

Octave indication.

```
>>> pitchtools.NumberedPitch(13).octave
Octave(5)
```

Returns octave.

`NumberedPitch.octave_number`

Octave number.

```
>>> pitchtools.NumberedPitch(13).octave_number
5
```

Returns integer.

`NumberedPitch.pitch_class_name`

Pitch-class name.

```
>>> pitchtools.NumberedPitch(13).pitch_class_name
'cs'
```

Returns string.

`NumberedPitch.pitch_class_number`  
Pitch-class number.

```
>>> pitchtools.NumberedPitch(13).pitch_class_number
1
```

Returns integer or float.

`NumberedPitch.pitch_class_octave_label`  
Pitch-class / octave label.

```
>>> pitchtools.NumberedPitch(13).pitch_class_octave_label
'C#5'
```

Returns string.

`NumberedPitch.pitch_name`  
Pitch name.

```
>>> pitchtools.NumberedPitch(13).pitch_name
'CS'''
```

Returns string.

`NumberedPitch.pitch_number`  
Pitch-class number.

```
>>> pitchtools.NumberedPitch(13).pitch_number
13
```

Returns integer or float.

## Methods

`NumberedPitch.apply_accidental (accidental=None)`  
Apply *accidental*.

```
>>> pitchtools.NumberedPitch(13).apply_accidental('flat')
NumberedPitch(12)
```

Emit new numbered pitch.

`NumberedPitch.invert (axis=None)`  
Invert around *axis*.

Not yet implemented.

`NumberedPitch.multiply (n=1)`  
Multiply pitch-class by *n*, maintaining octave.

```
>>> pitchtools.NumberedPitch(14).multiply(3)
NumberedPitch(18)
```

Emit new numbered pitch.

`NumberedPitch.transpose (n=0)`  
Transpose by *n* semitones.

```
>>> pitchtools.NumberedPitch(13).transpose(1)
NumberedPitch(14)
```

Emit new numbered pitch.

## Static methods

`(Pitch).is_diatonic_pitch_name (expr)`  
True when *expr* is a diatonic pitch name, otherwise false.

```
>>> pitchtools.Pitch.is_diatonic_pitch_name("c' ")
True
```

The regex `^[a-g,A-G])(, + | ' + | )$` underlies this predicate.

Returns boolean.

(Pitch) **.is\_diatonic\_pitch\_number** (*expr*)  
True when *expr* is a diatonic pitch number, otherwise false.

```
>>> pitchtools.Pitch.is_diatonic_pitch_number(7)
True
```

The diatonic pitch numbers are equal to the set of integers.

Returns boolean.

(Pitch) **.is\_pitch\_carrier** (*expr*)  
True when *expr* is an Abjad pitch, note, note-head of chord instance, otherwise false.

```
>>> note = Note("c'4")
>>> pitchtools.Pitch.is_pitch_carrier(note)
True
```

Returns boolean.

(Pitch) **.is\_pitch\_class\_octave\_number\_string** (*expr*)  
True when *expr* is a pitch-class / octave number string, otherwise false:

```
>>> pitchtools.Pitch.is_pitch_class_octave_number_string('C#2')
True
```

Quarternote accidentals are supported.

The regex `^([A-G])([#]{1,2}|[b]{1,2}|[#]?[+]|[b]?[~]|)([-]?[0-9]+)$` underlies this predicate.

Returns boolean.

(Pitch) **.is\_pitch\_name** (*expr*)  
True *expr* is a pitch name, otherwise false.

```
>>> pitchtools.Pitch.is_pitch_name('c,')
True
```

The regex `^([a-g,A-G])(([s]{1,2}|[f]{1,2}|t?q?[f,s]|)!)?(, + | ' + | )$` underlies this predicate.

Returns boolean.

(Pitch) **.is\_pitch\_number** (*expr*)  
True *expr* is a pitch number, otherwise false.

```
>>> pitchtools.Pitch.is_pitch_number(13)
True
```

The pitch numbers are equal to the set of all integers in union with the set of all integers plus of minus 0.5.

Returns boolean.

## Special methods

NumberedPitch.**\_\_abs\_\_** ()

NumberedPitch.**\_\_add\_\_** (*arg*)

NumberedPitch.**\_\_eq\_\_** (*expr*)

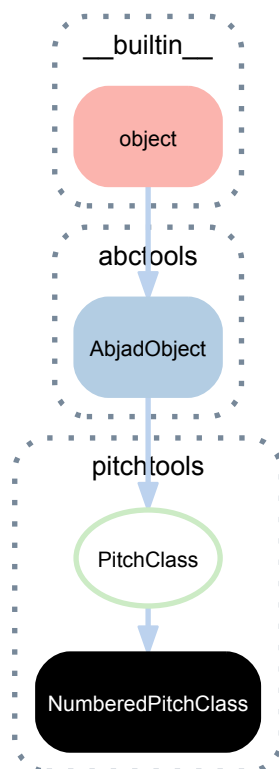
NumberedPitch.**\_\_float\_\_** ()

```

NumberedPitch.__ge__(other)
    x.__ge__(y) <==> x>=y
NumberedPitch.__gt__(other)
    x.__gt__(y) <==> x>y
NumberedPitch.__hash__()
NumberedPitch.__int__()
NumberedPitch.__le__(other)
    x.__le__(y) <==> x<=y
NumberedPitch.__lt__(expr)
    (AbjadObject).__ne__(expr)
    True when ID of expr does not equal ID of Abjad object.
    Returns boolean.
NumberedPitch.__neg__()
NumberedPitch.__repr__()
NumberedPitch.__str__()
NumberedPitch.__sub__(arg)

```

### 20.2.17 pitchtools.NumberedPitchClass



```

class pitchtools.NumberedPitchClass(expr)
    A numbered pitch-class.

```

```

>>> pitchtools.NumberedPitchClass(13)
NumberedPitchClass(1)

```

```

>>> pitchtools.NumberedPitchClass('d')
NumberedPitchClass(2)

```

```
>>> pitchtools.NumberedPitchClass(pitchtools.NamedPitch('g'))
NumberedPitchClass(7)
```

```
>>> pitchtools.NumberedPitchClass(pitchtools.NumberedPitch(15))
NumberedPitchClass(3)
```

```
>>> pitchtools.NumberedPitchClass(pitchtools.NamedPitchClass('e'))
NumberedPitchClass(4)
```

```
>>> pitchtools.NumberedPitchClass('C#5')
NumberedPitchClass(1)
```

```
>>> pitchtools.NumberedPitchClass(Note("a'8."))
NumberedPitchClass(9)
```

Returns numbered pitch-class.

## Bases

- `pitchtools.PitchClass`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`NumberedPitchClass.accidental`  
Accidental.

```
>>> pitchtools.NumberedPitchClass(1).accidental
Accidental('s')
```

Returns accidental.

`(PitchClass).accidental_spelling`  
Accidental spelling.

```
>>> pitchtools.NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

`NumberedPitchClass.alteration_in_semitones`  
Alteration in semitones.

```
>>> pitchtools.NumberedPitchClass(1).alteration_in_semitones
1
```

```
>>> pitchtools.NumberedPitchClass(10.5).alteration_in_semitones
-0.5
```

Returns integer or float.

`NumberedPitchClass.diatonic_pitch_class_name`  
Diatonic pitch-class name.

```
>>> pitchtools.NumberedPitchClass(1).diatonic_pitch_class_name
'c'
```

Returns string.

`NumberedPitchClass.diatonic_pitch_class_number`  
Diatonic pitch-class number.

```
>>> pitchtools.NumberedPitchClass(1).diatonic_pitch_class_number
0
```

Returns integer.

`NumberedPitchClass.named_pitch_class`  
Named pitch-class.

```
>>> pitchtools.NumberedPitchClass(13).named_pitch_class
NamedPitchClass('cs')
```

Returns named pitch-class.

`NumberedPitchClass.numbered_pitch_class`  
Numbered pitch-class.

```
>>> pitchtools.NumberedPitchClass(13).numbered_pitch_class
NumberedPitchClass(1)
```

Returns numbered pitch-class.

`NumberedPitchClass.pitch_class_label`  
Pitch-class / octave label.

```
>>> pitchtools.NumberedPitchClass(13).pitch_class_label
'C#'
```

Returns string.

`NumberedPitchClass.pitch_class_name`  
Pitch-class name.

```
>>> pitchtools.NumberedPitchClass(1).pitch_class_name
'cs'
```

Returns string.

`NumberedPitchClass.pitch_class_number`  
Pitch-class number.

```
>>> pitchtools.NumberedPitchClass(1).pitch_class_number
1
```

Returns integer or float.

## Methods

`NumberedPitchClass.apply_accidental(accidental=None)`  
Apply *accidental*.

```
>>> pitchtools.NumberedPitchClass(1).apply_accidental('flat')
NumberedPitchClass(0)
```

Emit new numbered pitch-class.

`NumberedPitchClass.invert()`  
Invert pitch-class.

Emit new numbered pitch-class.

`NumberedPitchClass.multiply(n=1)`  
Multiply pitch-class number by *n*:

```
>>> pitchtools.NumberedPitchClass(11).multiply(3)
NumberedPitchClass(9)
```

Emit new numbered pitch-class.

`NumberedPitchClass.transpose(n)`  
Transpose pitch-class by *n*.  
Emit new numbered pitch-class.

## Static methods

`(PitchClass).is_diatonic_pitch_class_name(expr)`  
True when *expr* is a diatonic pitch-class name, otherwise false.

```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_name('c')
True
```

The regex `^[a-g,A-G]$` underlies this predicate.

Returns boolean.

`(PitchClass).is_diatonic_pitch_class_number(expr)`  
True when *expr* is a diatonic pitch-class number, otherwise false.

```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_number(0)
True
```

```
>>> pitchtools.PitchClass.is_diatonic_pitch_class_number(-5)
False
```

The diatonic pitch-class numbers are equal to the set `[0, 1, 2, 3, 4, 5, 6]`.

Returns boolean.

`(PitchClass).is_pitch_class_name(expr)`  
True when *expr* is a pitch-class name, otherwise false.

```
>>> pitchtools.PitchClass.is_pitch_class_name('fs')
True
```

The regex `^([a-g,A-G])((([s]{1,2}|[f]{1,2}|t?q?[fs]|)!)?)$` underlies this predicate.

Returns boolean.

`(PitchClass).is_pitch_class_number(expr)`  
True *expr* is a pitch-class number, otherwise false.

```
>>> pitchtools.PitchClass.is_pitch_class_number(1)
True
```

The pitch-class numbers are equal to the set `[0, 0.5, ..., 11, 11.5]`.

Returns boolean.

## Special methods

`NumberedPitchClass.__abs__()`

`NumberedPitchClass.__add__(expr)`  
Addition defined against numbered intervals only.

`NumberedPitchClass.__copy__(*args)`

`NumberedPitchClass.__eq__(expr)`

`NumberedPitchClass.__float__()`

`(PitchClass).__hash__()`

`NumberedPitchClass.__int__()`



(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

NumberedPitchClass.**\_\_neg\_\_**()

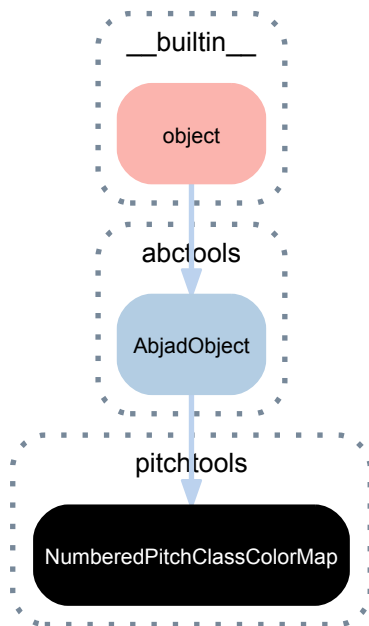
NumberedPitchClass.**\_\_repr\_\_**()

NumberedPitchClass.**\_\_str\_\_**()

NumberedPitchClass.**\_\_sub\_\_**(*expr*)

Subtraction defined against both numbered intervals and against other pitch-classes.

## 20.2.18 pitchtools.NumberedPitchClassColorMap



**class** pitchtools.**NumberedPitchClassColorMap**(*pitch\_iterables*, *colors*)

A numbered pitch-class color map.

```

>>> pitch_class_numbers = ... [[-8, 2, 10, 21], [0, 11, 32, 41], [15, 25, 42, 43]]
>>> colors = ['red', 'green', 'blue']
>>> mapping = pitchtools.NumberedPitchClassColorMap(
... pitch_class_numbers, colors)
  
```

Numbered pitch-class color maps are immutable.

### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

NumberedPitchClassColorMap.**colors**

NumberedPitchClassColorMap.**pairs**

NumberedPitchClassColorMap.**pitch\_iterables**

NumberedPitchClassColorMap.**twelve\_tone\_complete**

NumberedPitchClassColorMap.**twenty\_four\_tone\_complete**

## Methods

NumberedPitchClassColorMap.**get** (*key*, *alternative=None*)

## Special methods

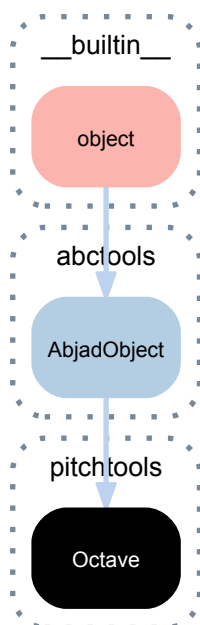
(AbjadObject).**\_\_eq\_\_** (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

NumberedPitchClassColorMap.**\_\_getitem\_\_** (*pc*)

(AbjadObject).**\_\_ne\_\_** (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

NumberedPitchClassColorMap.**\_\_repr\_\_** ()

### 20.2.19 pitchtools.Octave



**class** pitchtools.**Octave** (*expr*)  
An octave.

```
>>> pitchtools.Octave(4)
Octave(4)
```

```
>>> pitchtools.Octave(",", ",")
Octave(1)
```

```
>>> pitchtools.Octave(pitchtools.NamedPitch("cs'"))
Octave(5)
```

```
>>> pitchtools.Octave(pitchtools.Octave(2))
Octave(2)
```

Returns octave.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

Octave.**octave\_number**

Octave number of octave.

```
>>> pitchtools.Octave(5).octave_number
5
```

Returns integer.

Octave.**octave\_tick\_string**

LilyPond octave tick representation of octave.

```
>>> for i in range(-1, 9):
...     print i, pitchtools.Octave(i).octave_tick_string
-1 ',,,,'
0 ',,,
1 ',,
2 ',
3 '
4 '
5 ''
6 ''',
7 ''',,
8 ''',,,
```

Returns string.

Octave.**pitch\_number**

Pitch number of first note in octave.

```
>>> pitchtools.Octave(4).pitch_number
0
```

```
>>> pitchtools.Octave(5).pitch_number
12
```

```
>>> pitchtools.Octave(3).pitch_number
-12
```

Returns integer.

Octave.**pitch\_range**

Pitch range of octave.

```
>>> pitchtools.Octave(5).pitch_range
PitchRange(' [C5, C6]')
```

Returns pitch range.

## Class methods

Octave.**from\_pitch\_name** (*pitch\_name*)

Change *pitch\_name* to octave.

```
>>> pitchtools.Octave.from_pitch_name('cs')
Octave(3)
```

Returns integer.

Octave.**from\_pitch\_number**(*pitch\_number*)

Change *pitch\_number* to octave.

```
>>> pitchtools.Octave.from_pitch_number(13)
Octave(5)
```

Returns octave.

Octave.**is\_octave\_tick\_string**(*expr*)

True when *expr* is an octave tick string, otherwise false.

```
>>> pitchtools.Octave.is_octave_tick_string(',,,')
True
```

The regex `^, +|' +| $` underlies this predicate.

Returns boolean.

## Special methods

Octave.**\_\_eq\_\_**(*other*)

True if *other* is octave with same octave number, otherwise False.

```
>>> octave = pitchtools.Octave(4)
>>> octave == pitchtools.Octave(4)
True
```

```
>>> octave == pitchtools.Octave(3)
False
```

```
>>> octave == 'foo'
False
```

Returns boolean.

Octave.**\_\_float\_\_**()

Cast octave as floating-point number.

```
>>> float(pitchtools.Octave(3))
3.0
```

Returns floating-point number.

Octave.**\_\_hash\_\_**()

Octave.**\_\_int\_\_**()

Cast octave as integer.

```
>>> int(pitchtools.Octave(3))
3
```

Returns integer.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

Octave.**\_\_repr\_\_**()

Octave.**\_\_str\_\_**()

LilyPond octave tick representation of octave.

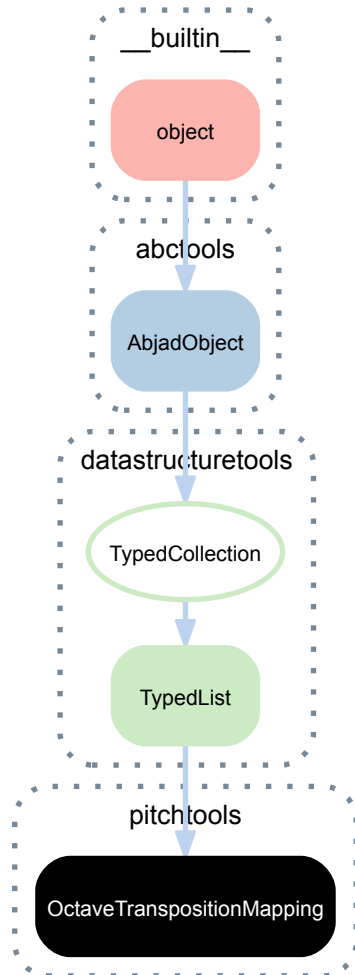
```
>>> str(pitchtools.Octave(4))
""
```

```
>>> str(pitchtools.Octave(1))
','
```

```
>>> str(pitchtools.Octave(3))
''
```

Returns string.

### 20.2.20 pitchtools.OctaveTranspositionMapping



**class** `pitchtools.OctaveTranspositionMapping` (*tokens=None, item\_class=None, name=None*)

An octave transposition mapping.

```
>>> mapping = pitchtools.OctaveTranspositionMapping(
...     [('A0, C4)', 15), ('[C4, C8)', 27)])
```

```
>>> mapping
OctaveTranspositionMapping([('A0, C4)', 15), ('[C4, C8)', 27)])
```

Octave transposition mappings model `pitchtools.transpose_pitch_number_by_octave_transposition_input`.

Octave transposition mappings implement the list interface and are mutable.

#### Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(TypedCollection) **.item\_class**

Item class to coerce tokens into.

OctaveTranspositionMapping **.storage\_format**

Octave transposition mapping storage format.

```
>>> print mapping.storage_format
pitchtools.OctaveTranspositionMapping([
  pitchtools.OctaveTranspositionMappingComponent(
    pitchtools.PitchRange(
      '[A0, C4]'
    ),
    pitchtools.NumberedPitch(15)
  ),
  pitchtools.OctaveTranspositionMappingComponent(
    pitchtools.PitchRange(
      '[C4, C8]'
    ),
    pitchtools.NumberedPitch(27)
  )
])
```

Returns string.

## Read/write properties

(TypedCollection) **.name**

Read / write name of typed tuple.

## Methods

(TypedList) **.append** (*token*)

Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

(TypedList) **.count** (*token*)

Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)
3
```

Returns count.

(TypedList) **.extend** (*tokens*)

Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

(TypedList) **.index** (*token*)

Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('cquf', 'as', 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as")
1
```

Returns index.

(TypedList) **.insert** (*i*, *token*)

Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('1', 2, 4.3))
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None*, *item\_class=None*, *name=None*)

(TypedList) **.pop** (*i=-1*)

Aliases list.pop().

(TypedList) **.remove** (*token*)

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()

Aliases list.reverse().

(TypedList) **.sort** (*cmp=None*, *key=None*, *reverse=False*)

Aliases list.sort().

## Special methods

`OctaveTranspositionMapping.__call__` (*pitches*)  
Call octave transposition mapping on *pitches*.

```
>>> mapping([-24, -22, -23, -21])  
[24, 26, 25, 15]
```

```
>>> mapping([0, 2, 1, 3])  
[36, 38, 37, 27]
```

Returns list.

(TypedCollection).**\_\_contains\_\_** (*token*)

(TypedList).**\_\_delitem\_\_** (*i*)  
Aliases list.**\_\_delitem\_\_** ().

(TypedCollection).**\_\_eq\_\_** (*expr*)

(TypedList).**\_\_getitem\_\_** (*i*)  
Aliases list.**\_\_getitem\_\_** ().

(TypedList).**\_\_iadd\_\_** (*expr*)  
Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(  
...     item_class=contexttools.DynamicMark)  
>>> dynamic_collection.append('ppp')  
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format  
datastructuretools.TypedList([  
    contexttools.DynamicMark(  
        'ppp',  
        target_context=stafftools.Staff  
    ),  
    contexttools.DynamicMark(  
        'p',  
        target_context=stafftools.Staff  
    ),  
    contexttools.DynamicMark(  
        'mp',  
        target_context=stafftools.Staff  
    ),  
    contexttools.DynamicMark(  
        'mf',  
        target_context=stafftools.Staff  
    ),  
    contexttools.DynamicMark(  
        'fff',  
        target_context=stafftools.Staff  
    )  
],  
    item_class=contexttools.DynamicMark  
)
```

Returns collection.

(TypedCollection).**\_\_iter\_\_** ()

(TypedCollection).**\_\_len\_\_** ()

(TypedCollection).**\_\_ne\_\_** (*expr*)

`OctaveTranspositionMapping.__repr__` ()

(TypedList).**\_\_reversed\_\_** ()  
Aliases list.**\_\_reversed\_\_** ().

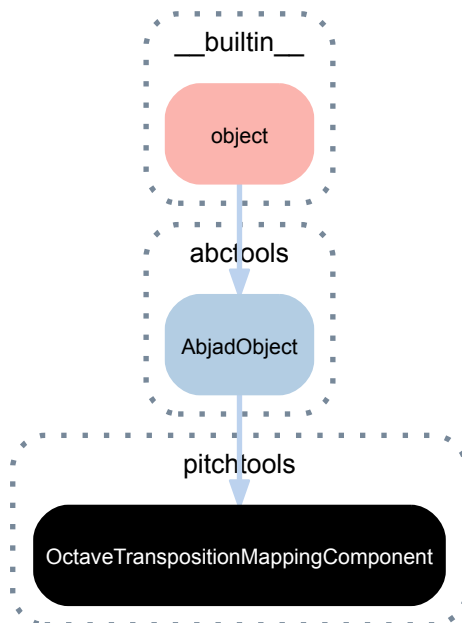


(`TypedList`).`__setitem__`(*i*, *expr*)  
 Change tokens in *expr* to items and set:

```
>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch('gqs,')
],
  item_class=pitchtools.NamedPitch
)
```

```
>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch("f'"),
  pitchtools.NamedPitch("g'"),
  pitchtools.NamedPitch("a'"),
  pitchtools.NamedPitch("b'"),
  pitchtools.NamedPitch("c'")
],
  item_class=pitchtools.NamedPitch
)
```

### 20.2.21 pitchtools.OctaveTranspositionMappingComponent



**class** `pitchtools.OctaveTranspositionMappingComponent` (\*args)  
 An octave transposition mapping component.

```
>>> mc = pitchtools.OctaveTranspositionMappingComponent(' [A0, C8]', 15)
>>> mc
OctaveTranspositionMappingComponent(' [A0, C8]', 15)
```

Initialize from input parameters separately, from a pair, from a string or from another mapping component.

**Model** `pitchtools.transpose_pitch_number_by_octave_transposition_mapping` input part. (See the docs for that function.)

Octave transposition mapping components are mutable.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read/write properties

`OctaveTranspositionMappingComponent.source_pitch_range`

Read / write source pitch range:

```
>>> mc.source_pitch_range
PitchRange('A0, C8')
```

Returns pitch range or none.

`OctaveTranspositionMappingComponent.target_octave_start_pitch`

Read / write target octave start pitch:

```
>>> mc.target_octave_start_pitch
NumberedPitch(15)
```

Returns numbered pitch or none.

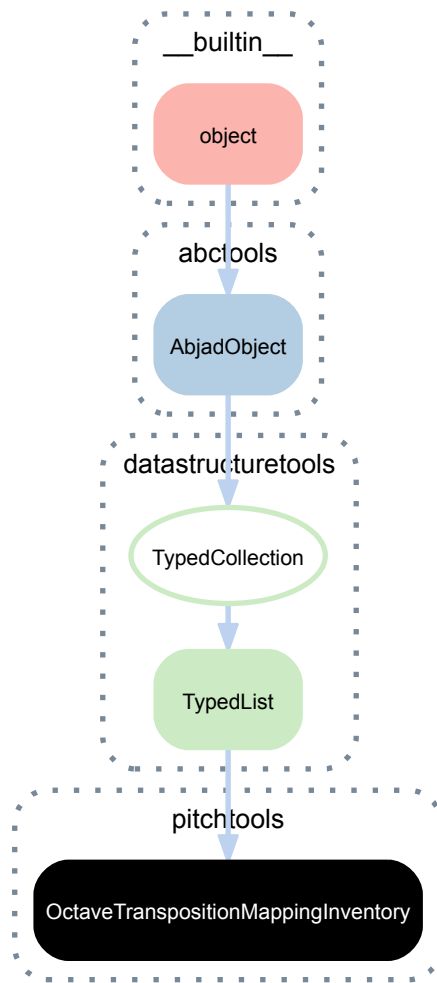
## Special methods

`OctaveTranspositionMappingComponent.__eq__(expr)`

`OctaveTranspositionMappingComponent.__ne__(expr)`

`OctaveTranspositionMappingComponent.__repr__()`

## 20.2.22 pitchtools.OctaveTranspositionMappingInventory



**class** `pitchtools.OctaveTranspositionMappingInventory` (*tokens=None*,  
*item\_class=None*,  
*name=None*)

An ordered list of octave transposition mappings.

```
>>> mapping_1 = pitchtools.OctaveTranspositionMapping(
...     [('A0, C4)', 15], ('C4, C8)', 27))
>>> mapping_2 = pitchtools.OctaveTranspositionMapping(
...     [('A0, C8)', -18])
>>> inventory = pitchtools.OctaveTranspositionMappingInventory(
...     [mapping_1, mapping_2])
```

```
>>> print inventory.storage_format
pitchtools.OctaveTranspositionMappingInventory([
  pitchtools.OctaveTranspositionMapping([
    pitchtools.OctaveTranspositionMappingComponent(
      pitchtools.PitchRange(
        'A0, C4)'
      ),
      pitchtools.NumberedPitch(15)
    ),
    pitchtools.OctaveTranspositionMappingComponent(
      pitchtools.PitchRange(
        'C4, C8)'
      ),
      pitchtools.NumberedPitch(27)
    )
  ]),
  pitchtools.OctaveTranspositionMapping([
    pitchtools.OctaveTranspositionMappingComponent(
```

```

        pitchtools.PitchRange(
            '[A0, C8]'
        ),
        pitchtools.NumberedPitch(-18)
    )
]
])

```

Octave transposition mapping inventories implement list interface and are mutable.

## Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TypedCollection).item_class`

Item class to coerce tokens into.

`(TypedCollection).storage_format`

Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`

Read / write name of typed tuple.

## Methods

`(TypedList).append(token)`

Change *token* to item and append:

```

>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]

```

```

>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]

```

Returns none.

`(TypedList).count(token)`

Change *token* to item and return count.

```

>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]

```

```

>>> integer_collection.count(0)
3

```

Returns count.

(TypedList) **.extend** (*tokens*)

Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

(TypedList) **.index** (*token*)

Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList (
...     tokens=('c'qf', "as'", 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

(TypedList) **.insert** (*i*, *token*)

Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('1', 2, 4.3))
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None*, *item\_class=None*, *name=None*)

(TypedList) **.pop** (*i=-1*)

Aliases list.pop().

(TypedList) **.remove** (*token*)

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()

Aliases list.reverse().

(TypedList) **.sort** (*cmp=None*, *key=None*, *reverse=False*)

Aliases list.sort().

## Special methods

(TypedCollection). **\_\_contains\_\_** (*token*)

(TypedList). **\_\_delitem\_\_** (*i*)  
Aliases list.**\_\_delitem\_\_** ().

(TypedCollection). **\_\_eq\_\_** (*expr*)

(TypedList). **\_\_getitem\_\_** (*i*)  
Aliases list.**\_\_getitem\_\_** ().

(TypedList). **\_\_iadd\_\_** (*expr*)  
Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList (
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList ([
    contexttools.DynamicMark (
        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark (
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark (
        'mp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark (
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark (
        'fff',
        target_context=stafftools.Staff
    )
],
    item_class=contexttools.DynamicMark
)
```

Returns collection.

(TypedCollection). **\_\_iter\_\_** ()

(TypedCollection). **\_\_len\_\_** ()

(TypedCollection). **\_\_ne\_\_** (*expr*)

(AbjadObject). **\_\_repr\_\_** ()  
Interpreter representation of Abjad object.

Returns string.

(TypedList). **\_\_reversed\_\_** ()  
Aliases list.**\_\_reversed\_\_** ().

(TypedList). **\_\_setitem\_\_** (*i*, *expr*)  
Change tokens in *expr* to items and set:

```
>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList ([
    pitchtools.NamedPitch ("c"),
    pitchtools.NamedPitch ("d"),
    pitchtools.NamedPitch ("e"),
    pitchtools.NamedPitch ('gqs,')
```

```

    ],
    item_class=pitchtools.NamedPitch
)

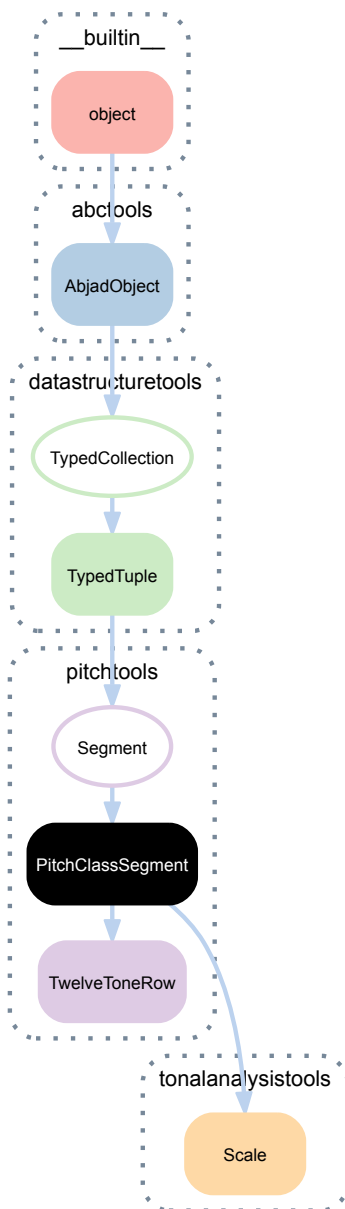
```

```

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch("f'"),
    pitchtools.NamedPitch("g'"),
    pitchtools.NamedPitch("a'"),
    pitchtools.NamedPitch("b'"),
    pitchtools.NamedPitch("c'")
],
    item_class=pitchtools.NamedPitch
)

```

### 20.2.23 pitchtools.PitchClassSegment



**class** `pitchtools.PitchClassSegment` (*tokens=None, item\_class=None, name=None*)  
A pitch-class segment.

```
>>> numbered_pitch_class_segment = pitchtools.PitchClassSegment(  
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],  
...     item_class=pitchtools.NumberedPitchClass,  
... )  
>>> numbered_pitch_class_segment  
PitchClassSegment([10, 10.5, 6, 7, 10.5, 7])
```

```
>>> named_pitch_class_segment = pitchtools.PitchClassSegment(  
...     tokens=['c', 'ef', 'bqs', 'd'],  
...     item_class=pitchtools.NamedPitchClass,  
... )  
>>> named_pitch_class_segment  
PitchClassSegment(['c', 'ef', 'bqs', 'd'])
```

Returns pitch-class segment.

## Bases

- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`PitchClassSegment.has_duplicates`  
True if segment contains duplicate items:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(  
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],  
... )  
>>> pitch_class_segment.has_duplicates  
True
```

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(  
...     tokens="c d e f g a b",  
... )  
>>> pitch_class_segment.has_duplicates  
False
```

Returns boolean.

(`TypedCollection`).**item\_class**  
Item class to coerce tokens into.

(`TypedCollection`).**storage\_format**  
Storage format of typed tuple.

## Read/write properties

(`TypedCollection`).**name**  
Read / write name of typed tuple.



## Methods

`PitchClassSegment.alpha()`

Morris alpha transform of pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.alpha()
PitchClassSegment([11, 11.5, 7, 6, 11.5, 6])
```

Emit new pitch-class segment.

`(TypedTuple).count(token)`

Change *token* to item and return count in collection.

`(TypedTuple).index(token)`

Change *token* to item and return index in collection.

`PitchClassSegment.invert()`

Invert pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.invert()
PitchClassSegment([2, 1.5, 6, 5, 1.5, 5])
```

Emit new pitch-class segment.

`PitchClassSegment.is_equivalent_under_transposition(expr)`

True if equivalent under transposition to *expr*, otherwise False.

Returns boolean.

`PitchClassSegment.make_notes(n=None, written_duration=None)`

Make first *n* notes in pitch class segment.

Set *n* equal to *n* or length of segment.

Set *written\_duration* equal to *written\_duration* or 1/8:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     [2, 4.5, 6, 11, 4.5, 10])
```

```
>>> notes = pitch_class_segment.make_notes()
>>> staff = Staff(notes)
>>> show(staff)
```



Allow nonassignable *written\_duration*:

```
>>> notes = pitch_class_segment.make_notes(4, Duration(5, 16))
>>> staff = Staff(notes)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))(staff)
>>> show(staff)
```



Returns list of notes.

`PitchClassSegment.multiply(n)`

Multiply pitch-class segment by *n*:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.multiply(5)
PitchClassSegment([2, 4.5, 6, 11, 4.5, 11])
```

Emit new pitch-class segment.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

PitchClassSegment **.retrograde** ()

Retrograde of pitch-class segment:

```
>>> pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).retrograde()
PitchClassSegment([7, 10.5, 7, 6, 10.5, 10])
```

Emit new pitch-class segment.

PitchClassSegment **.rotate** (*n*)

Rotate pitch-class segment:

```
>>> pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).rotate(1)
PitchClassSegment([7, 10, 10.5, 6, 7, 10.5])
```

```
>>> pitchtools.PitchClassSegment (
...     tokens=['c', 'ef', 'bqs', 'd'],
...     ).rotate(-2)
PitchClassSegment(['bqs', 'd', 'c', 'ef'])
```

Emit new pitch-class segment.

PitchClassSegment **.transpose** (*expr*)

Transpose pitch-class segment:

```
>>> pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).transpose(10)
PitchClassSegment([8, 8.5, 4, 5, 8.5, 5])
```

Emit new pitch-class segment.

## Class methods

PitchClassSegment **.from\_selection** (*selection, item\_class=None, name=None*)

Initialize pitch-class segment from component selection:

```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> pitchtools.PitchClassSegment.from_selection(selection)
PitchClassSegment(['c', 'd', 'fs', 'a', 'b', 'c', 'g'])
```

Returns pitch-class segment.

## Special methods

(TypedTuple) **.\_\_add\_\_** (*expr*)

(TypedTuple) **.\_\_contains\_\_** (*token*)

Change *token* to item and return true if item exists in collection.

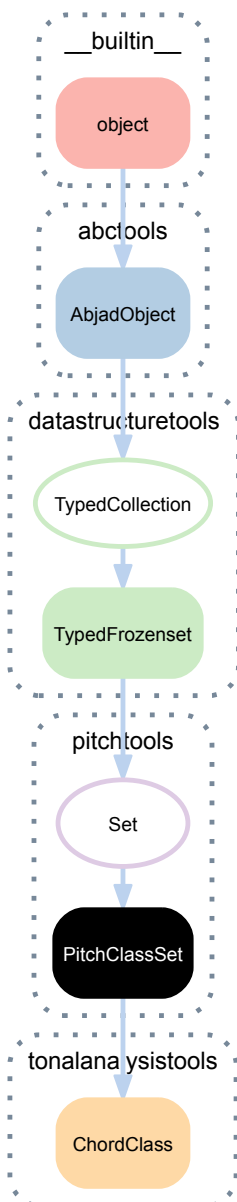
(TypedCollection) **.\_\_eq\_\_** (*expr*)

```

(TypedTuple) .__getitem__(i)
    Aliases tuple.__getitem__().
(TypedTuple) .__getslice__(start, stop)
(TypedTuple) .__hash__()
(TypedCollection) .__iter__()
(TypedCollection) .__len__()
(TypedTuple) .__mul__(expr)
(TypedCollection) .__ne__(expr)
(Segment) .__repr__()
(TypedTuple) .__rmul__(expr)
(Segment) .__str__()

```

### 20.2.24 pitchtools.PitchClassSet



**class** `pitchtools.PitchClassSet` (*tokens=None, item\_class=None, name=None*)

A pitch-class set.

```
>>> numbered_pitch_class_set = pitchtools.PitchClassSet(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     item_class=pitchtools.NumberedPitchClass,
... )
>>> numbered_pitch_class_set
PitchClassSet([6, 7, 10, 10.5])
```

```
>>> named_pitch_class_set = pitchtools.PitchClassSet(
...     tokens=['c', 'ef', 'bqs', 'd'],
...     item_class=pitchtools.NamedPitchClass,
... )
>>> named_pitch_class_set
PitchClassSet(['c', 'd', 'ef', 'bqs'])
```

Returns pitch-class set.

## Bases

- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(`TypedCollection`).**`item_class`**  
Item class to coerce tokens into.

(`TypedCollection`).**`storage_format`**  
Storage format of typed tuple.

## Read/write properties

(`TypedCollection`).**`name`**  
Read / write name of typed tuple.

## Methods

(`TypedFrozenSet`).**`copy()`**

(`TypedFrozenSet`).**`difference(expr)`**

(`TypedFrozenSet`).**`intersection(expr)`**

`PitchClassSet`.**`invert()`**  
Invert numbered pitch-class set:

```
>>> pitchtools.PitchClassSet(
...     [-2, -1.5, 6, 7, -1.5, 7],
... ).invert()
PitchClassSet([1.5, 2, 5, 6])
```

Returns numbered pitch-class set.

`PitchClassSet`.**`is_transposed_subset(pcset)`**  
True when self is transposed subset of *pcset*. False otherwise:

```
>>> pitch_class_set_1 = pitchtools.PitchClassSet(
...     [-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_set_2 = pitchtools.PitchClassSet(
...     [-2, -1.5, 6, 7, -1.5, 7, 7.5, 8],
...     )

>>> pitch_class_set_1.is_transposed_subset(pitch_class_set_2)
True
```

Returns boolean.

`PitchClassSet.is_transposed_superset(pcset)`  
 True when self is transposed superset of *pcset*. False otherwise:

```
>>> pitch_class_set_1 = pitchtools.PitchClassSet(
...     [-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_set_2 = pitchtools.PitchClassSet(
...     [-2, -1.5, 6, 7, -1.5, 7, 7.5, 8],
...     )

>>> pitch_class_set_2.is_transposed_superset(pitch_class_set_1)
True
```

Returns boolean.

(TypedFrozenset).`isdisjoint(expr)`

(TypedFrozenset).`issubset(expr)`

(TypedFrozenset).`issuperset(expr)`

`PitchClassSet.multiply(n)`  
 Multiply pitch-class set by *n*:

```
>>> pitchtools.PitchClassSet(
...     [-2, -1.5, 6, 7, -1.5, 7],
...     ).multiply(5)
PitchClassSet([2, 4.5, 6, 11])
```

Returns numbered pitch-class set.

(TypedCollection).`new(tokens=None, item_class=None, name=None)`

`PitchClassSet.order_by(pitch_class_segment)`

(TypedFrozenset).`symmetric_difference(expr)`

`PitchClassSet.transpose(expr)`  
 Transpose all pitch classes in self by *expr*.

(TypedFrozenset).`union(expr)`

## Class methods

`PitchClassSet.from_selection(selection, item_class=None, name=None)`  
 Initialize pitch-class set from component selection:

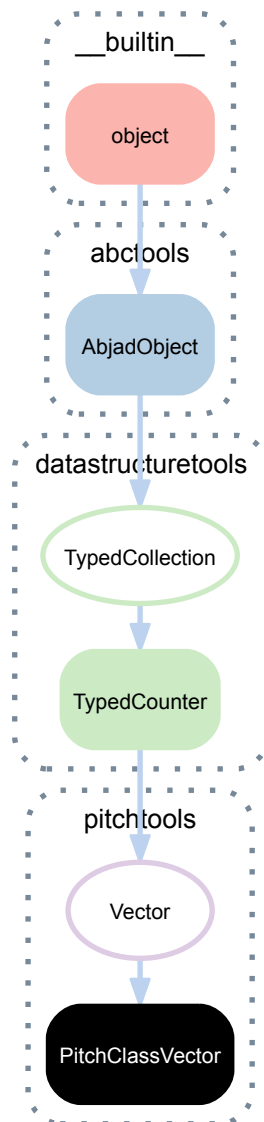
```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> pitchtools.PitchClassSet.from_selection(selection)
PitchClassSet(['c', 'd', 'fs', 'g', 'a', 'b'])
```

Returns pitch-class set.

## Special methods

```
(TypedFrozenset).__and__(expr)
(TypedCollection).__contains__(token)
(TypedCollection).__eq__(expr)
(TypedFrozenset).__ge__(expr)
(TypedFrozenset).__gt__(expr)
PitchClassSet.__hash__()
(TypedCollection).__iter__()
(TypedFrozenset).__le__(expr)
(TypedCollection).__len__()
(TypedFrozenset).__lt__(expr)
(TypedFrozenset).__ne__(expr)
(TypedFrozenset).__or__(expr)
(Set).__repr__()
(Set).__str__()
(TypedFrozenset).__sub__(expr)
(TypedFrozenset).__xor__(expr)
```

### 20.2.25 pitchtools.PitchClassVector



**class** `pitchtools.PitchClassVector` (*tokens=None, item\_class=None, name=None*)  
 A pitch-class vector.

#### Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(TypedCollection).item_class`  
 Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

### Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

### Methods

`(TypedCounter).clear()`  
`(TypedCounter).copy()`  
`(TypedCounter).elements()`  
`(TypedCounter).items()`  
`(TypedCounter).iteritems()`  
`(TypedCounter).iterkeys()`  
`(TypedCounter).intervalvalues()`  
`(TypedCounter).keys()`  
`(TypedCounter).most_common(n=None)`  
`(TypedCollection).new(tokens=None, item_class=None, name=None)`  
`(TypedCounter).subtract(iterable=None, **kwargs)`  
`(TypedCounter).update(iterable=None, **kwargs)`  
`(TypedCounter).values()`  
`(TypedCounter).viewitems()`  
`(TypedCounter).viewkeys()`  
`(TypedCounter).viewvalues()`

### Class methods

`PitchClassVector.from_selection(selection, item_class=None, name=None)`

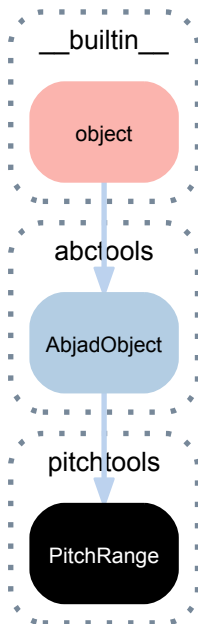
### Special methods

`(TypedCounter).__add__(expr)`  
`(TypedCounter).__and__(expr)`  
`(TypedCollection).__contains__(token)`  
`(TypedCounter).__delitem__(token)`  
`(TypedCollection).__eq__(expr)`  
`(TypedCounter).__getitem__(token)`  
`(TypedCollection).__iter__()`  
`(TypedCollection).__len__()`  
`(TypedCounter).__missing__(token)`  
`(TypedCollection).__ne__(expr)`



```
(TypedCounter).__or__(expr)
(Vector).__repr__()
(TypedCounter).__setitem__(token, value)
(Vector).__str__()
(TypedCounter).__sub__(expr)
```

## 20.2.26 pitchtools.PitchRange



```
class pitchtools.PitchRange(*args, **kwargs)
    A pitch range.
```

```
>>> pitch_range = pitchtools.PitchRange(-12, 36,
...     pitch_range_name='four-octave range')
>>> print pitch_range.storage_format
pitchtools.PitchRange(
    '[C3, C7]',
    pitch_range_name='four-octave range',
    pitch_range_name_markup=markuptools.Markup((
        'four-octave range',
    ))
)
```

Initialize from pitch numbers, pitch names, pitch instances, one-line reprs or other pitch range objects.

Pitch ranges implement equality testing against other pitch ranges.

Pitch ranges test less than, greater than, less-equal and greater-equal against pitches.

Pitch ranges do not sort relative to other pitch ranges.

Pitch ranges are immutable.

### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`PitchRange.one_line_named_pitch_repr`

One-line named pitch repr of pitch of range:

```
>>> pitch_range.one_line_named_pitch_repr
'[C3, C7]'
```

Returns string.

`PitchRange.one_line_numbered_pitch_repr`

One-line numbered pitch repr of pitch of range:

```
>>> pitch_range.one_line_numbered_pitch_repr
'[-12, 36]'
```

Returns string.

`PitchRange.pitch_range_name`

Name of pitch range:

```
>>> pitch_range.pitch_range_name
'four-octave range'
```

Returns string or none.

`PitchRange.pitch_range_name_markup`

Markup of pitch range name:

```
>>> pitch_range.pitch_range_name_markup
Markup(('four-octave range',))
```

Default to `pitch_range_name` when `pitch_range_name_markup` not set explicitly.

Returns markup or none.

`PitchRange.start_pitch`

Start pitch of range:

```
>>> pitch_range.start_pitch
NamedPitch('c')
```

Returns pitch.

`PitchRange.start_pitch_is_included_in_range`

Boolean true when start pitch is included in range. Otherwise false:

```
>>> pitch_range.start_pitch_is_included_in_range
True
```

Returns boolean.

`PitchRange.stop_pitch`

Stop pitch of range:

```
>>> pitch_range.stop_pitch
NamedPitch("c'")
```

Returns pitch.

`PitchRange.stop_pitch_is_included_in_range`

Boolean true when stop pitch is included in range. Otherwise false:

```
>>> pitch_range.stop_pitch_is_included_in_range
True
```

Returns boolean.

`PitchRange.storage_format`

Storage format of pitch range.

Returns string.

## Class methods

`PitchRange.is_symbolic_pitch_range_string(expr)`

True when *expr* is a symbolic pitch range string. Otherwise false:

```
>>> pitchtools.PitchRange.is_symbolic_pitch_range_string(  
...     '[A0, C8]')  
True
```

The regex that underlies this predicate matches against two comma-separated pitch indicators enclosed in some combination of square brackets and round parentheses.

Returns boolean.

## Special methods

`PitchRange.__contains__(arg)`

`PitchRange.__eq__(arg)`

`PitchRange.__ge__(arg)`

`PitchRange.__gt__(arg)`

`PitchRange.__le__(arg)`

`PitchRange.__lt__(arg)`

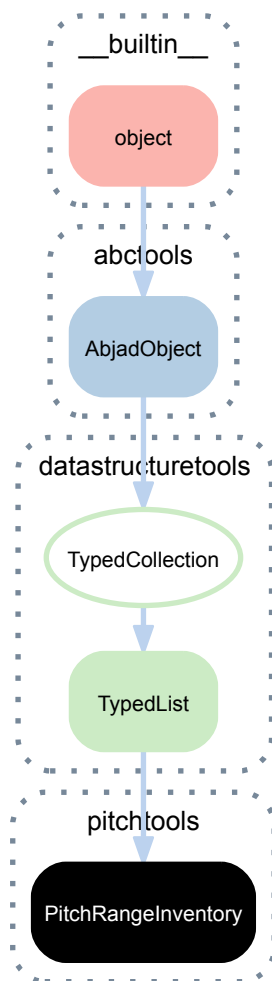
`PitchRange.__ne__(arg)`

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 20.2.27 `pitchtools.PitchRangeInventory`



**class** `pitchtools.PitchRangeInventory` (*tokens=None, item\_class=None, name=None*)  
 An ordered list of pitch ranges.

```
>>> pitchtools.PitchRangeInventory(['[C3, C6]', '[C4, C6]'])
PitchRangeInventory([PitchRange('[C3, C6]'), PitchRange('[C4, C6]')])
```

Pitch range inventories implement list interface and are mutable.

### Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

(`TypedCollection`).**item\_class**  
 Item class to coerce tokens into.

(`TypedCollection`).**storage\_format**  
 Storage format of typed tuple.

## Read/write properties

(TypedCollection) **.name**  
Read / write name of typed tuple.

## Methods

(TypedList) **.append** (*token*)  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

(TypedList) **.count** (*token*)  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)
3
```

Returns count.

(TypedList) **.extend** (*tokens*)  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

(TypedList) **.index** (*token*)  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('c'qf', "as'", 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

(TypedList) **.insert** (*i*, *token*)  
Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('1', 2, 4.3))
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

(TypedList) **.pop** (*i=-1*)

Aliases list.pop().

(TypedList) **.remove** (*token*)

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()

Aliases list.reverse().

(TypedList) **.sort** (*cmp=None, key=None, reverse=False*)

Aliases list.sort().

## Special methods

(TypedCollection) **.\_\_contains\_\_** (*token*)

(TypedList) **.\_\_delitem\_\_** (*i*)

Aliases list.\_\_delitem\_\_().

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedList) **.\_\_getitem\_\_** (*i*)

Aliases list.\_\_getitem\_\_().

(TypedList) **.\_\_iadd\_\_** (*expr*)

Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mp',
        target_context=stafftools.Staff
    )
])
```

```

    ),
    contexttools.DynamicMark(
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'fff',
        target_context=stafftools.Staff
    )
],
item_class=contexttools.DynamicMark
)

```

Returns collection.

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

(TypedCollection).**\_\_ne\_\_**(*expr*)

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList).**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList).**\_\_setitem\_\_**(*i*, *expr*)

Change tokens in *expr* to items and set:

```

>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch('gqs,')
],
item_class=pitchtools.NamedPitch
)

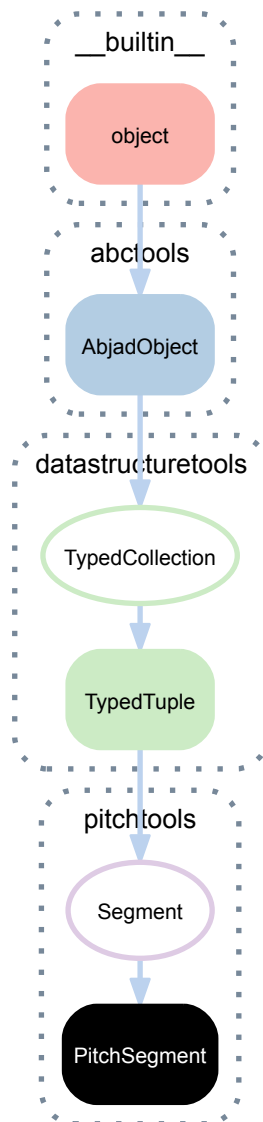
```

```

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch("f'"),
    pitchtools.NamedPitch("g'"),
    pitchtools.NamedPitch("a'"),
    pitchtools.NamedPitch("b'"),
    pitchtools.NamedPitch("c'")
],
item_class=pitchtools.NamedPitch
)

```

## 20.2.28 pitchtools.PitchSegment



**class** `pitchtools.PitchSegment` (*tokens=None, item\_class=None, name=None*)  
 A pitch segment.

```
>>> numbered_pitch_segment = pitchtools.PitchSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     item_class=pitchtools.NumberedPitch,
... )
>>> numbered_pitch_segment
PitchSegment([-2, -1.5, 6, 7, -1.5, 7])
```

```
>>> named_pitch_segment = pitchtools.PitchSegment(
...     ['bf', 'aq', "fs", "g", "bqf", "g"],
...     item_class=pitchtools.NamedPitch,
... )
>>> named_pitch_segment
PitchSegment(['bf', 'aq', "fs", "g", "bqf", "g"])
```

Returns pitch segment.

### Bases

- `pitchtools.Segment`



- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`PitchSegment.has_duplicates`

True if segment contains duplicate items:

```
>>> pitch_class_segment = pitchtools.PitchSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.has_duplicates
True
```

```
>>> pitch_class_segment = pitchtools.PitchSegment(
...     tokens="c d e f g a b",
...     )
>>> pitch_class_segment.has_duplicates
False
```

Returns boolean.

`PitchSegment.inflection_point_count`

`(TypedCollection).item_class`

Item class to coerce tokens into.

`PitchSegment.local_maxima`

`PitchSegment.local_minima`

`(TypedCollection).storage_format`

Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`

Read / write name of typed tuple.

## Methods

`(TypedTuple).count(token)`

Change *token* to item and return count in collection.

`(TypedTuple).index(token)`

Change *token* to item and return index in collection.

`PitchSegment.invert(axis)`

Invert pitch segment around *axis*.

Emit new pitch segment.

`PitchSegment.is_equivalent_under_transposition(expr)`

True if equivalent under transposition to *expr*, otherwise False.

Returns boolean.

`PitchSegment.make_notes(n=None, written_duration=None)`

Make first *n* notes in pitch class segment.

Set *n* equal to *n* or length of segment.

Set *written\_duration* equal to *written\_duration* or 1/8:

```
>>> notes = named_pitch_segment.make_notes()
>>> staff = Staff(notes)
>>> show(staff)
```



Allow nonassignable *written\_duration*:

```
>>> notes = named_pitch_segment.make_notes(4, Duration(5, 16))
>>> staff = Staff(notes)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))(staff)
>>> show(staff)
```



Returns list of notes.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

PitchSegment **.retrograde** ()

Retrograde of pitch segment:

```
>>> named_pitch_segment.retrograde()
PitchSegment(["g'", 'bqf', "g'", "fs'", 'aqs', 'bf,'])
```

Emit new pitch segment.

PitchSegment **.rotate** (*n*)

Rotate pitch segment:

```
>>> numbered_pitch_segment.rotate(1)
PitchSegment([7, -2, -1.5, 6, 7, -1.5])
```

```
>>> named_pitch_segment.rotate(-2)
PitchSegment(["fs'", "g'", 'bqf', "g'", 'bf,', 'aqs'])
```

Emit new pitch segment.

PitchSegment **.transpose** (*expr*)

Transpose pitch segment by *expr*.

Emit new pitch segment.

## Class methods

PitchSegment **.from\_selection** (*selection, item\_class=None, name=None*)

Initialize pitch segment from component selection:

```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> pitchtools.PitchSegment.from_selection(selection)
PitchSegment(["c'", "d'", "fs'", "a'", 'b', 'c', 'g'])
```

Returns pitch segment.

## Special methods

(TypedTuple) **.\_\_add\_\_** (*expr*)

(TypedTuple) .**\_\_contains\_\_** (*token*)  
Change *token* to item and return true if item exists in collection.

(TypedCollection) .**\_\_eq\_\_** (*expr*)

(TypedTuple) .**\_\_getitem\_\_** (*i*)  
Aliases tuple.**\_\_getitem\_\_** ().

(TypedTuple) .**\_\_getslice\_\_** (*start*, *stop*)

(TypedTuple) .**\_\_hash\_\_** ()

(TypedCollection) .**\_\_iter\_\_** ()

(TypedCollection) .**\_\_len\_\_** ()

(TypedTuple) .**\_\_mul\_\_** (*expr*)

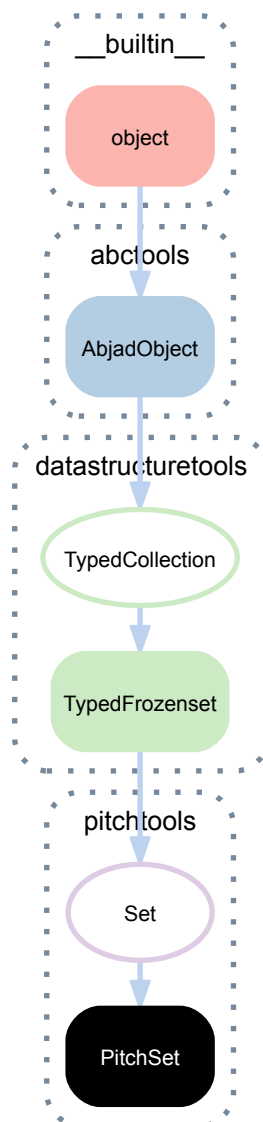
(TypedCollection) .**\_\_ne\_\_** (*expr*)

(Segment) .**\_\_repr\_\_** ()

(TypedTuple) .**\_\_rmul\_\_** (*expr*)

(Segment) .**\_\_str\_\_** ()

## 20.2.29 pitchtools.PitchSet



**class** `pitchtools.PitchSet` (*tokens=None, item\_class=None, name=None*)  
 A pitch segment.

```
>>> numbered_pitch_set = pitchtools.PitchSet(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     item_class=pitchtools.NumberedPitch,
... )
>>> numbered_pitch_set
PitchSet([-2, -1.5, 6, 7])
```

```
>>> named_pitch_set = pitchtools.PitchSet(
...     ['bf,', 'aqs', "fs'", "g'", 'bqf', "g'"],
...     item_class=pitchtools.NamedPitch,
... )
>>> named_pitch_set
PitchSet(['bf,', 'aqs', 'bqf', "fs'", "g'"])
```

Returns pitch segment.

### Bases

- `pitchtools.Set`

- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`PitchSet.duplicate_pitch_classes`

`PitchSet.is_pitch_class_unique`

`(TypedCollection).item_class`

Item class to coerce tokens into.

`(TypedCollection).storage_format`

Storage format of typed tuple.

### Read/write properties

`(TypedCollection).name`

Read / write name of typed tuple.

### Methods

`(TypedFrozenSet).copy()`

`(TypedFrozenSet).difference(expr)`

`(TypedFrozenSet).intersection(expr)`

`PitchSet.invert(axis)`

Invert pitch set around *axis*.

Emit new pitch set.

`PitchSet.is_equivalent_under_transposition(expr)`

True if equivalent under transposition to *expr*, otherwise False.

Returns boolean.

`(TypedFrozenSet).isdisjoint(expr)`

`(TypedFrozenSet).issubset(expr)`

`(TypedFrozenSet).issuperset(expr)`

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

`(TypedFrozenSet).symmetric_difference(expr)`

`PitchSet.transpose(expr)`

Transpose all pitches in self by *expr*.

`(TypedFrozenSet).union(expr)`

### Class methods

`PitchSet.from_selection(selection, item_class=None, name=None)`

Initialize pitch set from component selection:

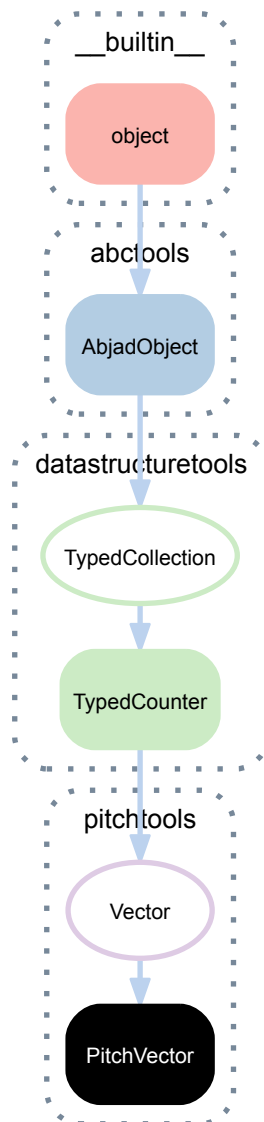
```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> pitchtools.PitchSet.from_selection(selection)
PitchSet(['c', 'g', 'b', 'c', 'd', 'fs', 'a'])
```

Returns pitch set.

### Special methods

```
(TypedFrozenSet).__and__(expr)
(TypedCollection).__contains__(token)
(TypedCollection).__eq__(expr)
(TypedFrozenSet).__ge__(expr)
(TypedFrozenSet).__gt__(expr)
(TypedFrozenSet).__hash__()
(TypedCollection).__iter__()
(TypedFrozenSet).__le__(expr)
(TypedCollection).__len__()
(TypedFrozenSet).__lt__(expr)
(TypedFrozenSet).__ne__(expr)
(TypedFrozenSet).__or__(expr)
(Set).__repr__()
(Set).__str__()
(TypedFrozenSet).__sub__(expr)
(TypedFrozenSet).__xor__(expr)
```

### 20.2.30 pitchtools.PitchVector



**class** `pitchtools.PitchVector` (*tokens=None, item\_class=None, name=None*)  
 A pitch vector.

#### Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(TypedCollection).item_class`  
 Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

### Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

### Methods

`(TypedCounter).clear()`  
`(TypedCounter).copy()`  
`(TypedCounter).elements()`  
`(TypedCounter).items()`  
`(TypedCounter).iteritems()`  
`(TypedCounter).iterkeys()`  
`(TypedCounter).intervalvalues()`  
`(TypedCounter).keys()`  
`(TypedCounter).most_common(n=None)`  
`(TypedCollection).new(tokens=None, item_class=None, name=None)`  
`(TypedCounter).subtract(iterable=None, **kwargs)`  
`(TypedCounter).update(iterable=None, **kwargs)`  
`(TypedCounter).values()`  
`(TypedCounter).viewitems()`  
`(TypedCounter).viewkeys()`  
`(TypedCounter).viewvalues()`

### Class methods

`PitchVector.from_selection(selection, item_class=None, name=None)`

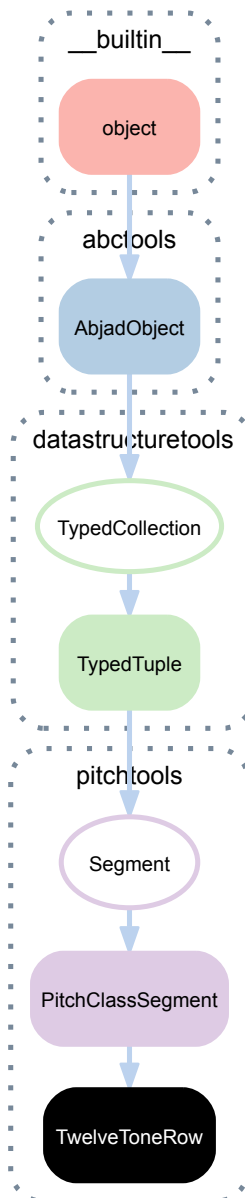
### Special methods

`(TypedCounter).__add__(expr)`  
`(TypedCounter).__and__(expr)`  
`(TypedCollection).__contains__(token)`  
`(TypedCounter).__delitem__(token)`  
`(TypedCollection).__eq__(expr)`  
`(TypedCounter).__getitem__(token)`  
`(TypedCollection).__iter__()`  
`(TypedCollection).__len__()`  
`(TypedCounter).__missing__(token)`  
`(TypedCollection).__ne__(expr)`



```
(TypedCounter).__or__(expr)
(Vector).__repr__()
(TypedCounter).__setitem__(token, value)
(Vector).__str__()
(TypedCounter).__sub__(expr)
```

### 20.2.31 pitchtools.TwelveToneRow



**class** `pitchtools.TwelveToneRow` (*tokens=None, name=None*)  
 A twelve-tone row.

```
>>> pitchtools.TwelveToneRow([0, 1, 11, 9, 3, 6, 7, 5, 4, 10, 2, 8])
TwelveToneRow([0, 1, 11, 9, 3, 6, 7, 5, 4, 10, 2, 8])
```

Twelve-tone rows validate pitch-classes at initialization.

Twelve-tone rows inherit canonical operators from numbered pitch-class segment.

Twelve-tone rows return numbered pitch-class segments on calls to `getslice`.

Twelve-tone rows are immutable.

## Bases

- `pitchtools.PitchClassSegment`
- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(PitchClassSegment).has_duplicates`

True if segment contains duplicate items:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.has_duplicates
True
```

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     tokens="c d e f g a b",
...     )
>>> pitch_class_segment.has_duplicates
False
```

Returns boolean.

`(TypedCollection).item_class`

Item class to coerce tokens into.

`(TypedCollection).storage_format`

Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`

Read / write name of typed tuple.

## Methods

`(PitchClassSegment).alpha()`

Morris alpha transform of pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.alpha()
PitchClassSegment([11, 11.5, 7, 6, 11.5, 6])
```

Emit new pitch-class segment.

`(TypedTuple).count(token)`

Change *token* to item and return count in collection.

`(TypedTuple).index(token)`

Change *token* to item and return index in collection.

`(PitchClassSegment).invert()`

Invert pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.invert()
PitchClassSegment([2, 1.5, 6, 5, 1.5, 5])
```

Emit new pitch-class segment.

`(PitchClassSegment).is_equivalent_under_transposition(expr)`

True if equivalent under transposition to *expr*, otherwise False.

Returns boolean.

`(PitchClassSegment).make_notes(n=None, written_duration=None)`

Make first *n* notes in pitch class segment.

Set *n* equal to *n* or length of segment.

Set *written\_duration* equal to *written\_duration* or 1/8:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     [2, 4.5, 6, 11, 4.5, 10])
```

```
>>> notes = pitch_class_segment.make_notes()
>>> staff = Staff(notes)
>>> show(staff)
```



Allow nonassignable *written\_duration*:

```
>>> notes = pitch_class_segment.make_notes(4, Duration(5, 16))
>>> staff = Staff(notes)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))(staff)
>>> show(staff)
```



Returns list of notes.

`(PitchClassSegment).multiply(n)`

Multiply pitch-class segment by *n*:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.multiply(5)
PitchClassSegment([2, 4.5, 6, 11, 4.5, 11])
```

Emit new pitch-class segment.

`TwelveToneRow.new(tokens=None, name=None)`

`(PitchClassSegment).retrograde()`

Retrograde of pitch-class segment:

```
>>> pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).retrograde()
PitchClassSegment([7, 10.5, 7, 6, 10.5, 10])
```

Emit new pitch-class segment.

`(PitchClassSegment).rotate(n)`

Rotate pitch-class segment:

```
>>> pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).rotate(1)
PitchClassSegment([7, 10, 10.5, 6, 7, 10.5])
```

```
>>> pitchtools.PitchClassSegment (
...     tokens=['c', 'ef', 'bqs', 'd'],
...     ).rotate(-2)
PitchClassSegment(['bqs', 'd', 'c', 'ef'])
```

Emit new pitch-class segment.

(PitchClassSegment).**transpose**(*expr*)

Transpose pitch-class segment:

```
>>> pitchtools.PitchClassSegment (
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).transpose(10)
PitchClassSegment([8, 8.5, 4, 5, 8.5, 5])
```

Emit new pitch-class segment.

## Class methods

TwelveToneRow.**from\_selection**(*selection*, *item\_class=None*, *name=None*)

## Special methods

(TypedTuple).**\_\_add\_\_**(*expr*)

(TypedTuple).**\_\_contains\_\_**(*token*)

Change *token* to item and return true if item exists in collection.

(TypedCollection).**\_\_eq\_\_**(*expr*)

(TypedTuple).**\_\_getitem\_\_**(*i*)

Aliases tuple.**\_\_getitem\_\_**().

TwelveToneRow.**\_\_getslice\_\_**(*start*, *stop*)

(TypedTuple).**\_\_hash\_\_**()

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

TwelveToneRow.**\_\_mul\_\_**(*expr*)

(TypedCollection).**\_\_ne\_\_**(*expr*)

(Segment).**\_\_repr\_\_**()

TwelveToneRow.**\_\_rmul\_\_**(*expr*)

(Segment).**\_\_str\_\_**()

## 20.3 Functions

### 20.3.1 pitchtools.apply\_accidental\_to\_named\_pitch

`pitchtools.apply_accidental_to_named_pitch`(*named\_pitch*, *accidental=None*)

Apply *accidental* to *named\_pitch*:

```
>>> pitch = pitchtools.NamedPitch("cs'")
>>> pitchtools.apply_accidental_to_named_pitch(pitch, 'f')
NamedPitch("c'")
```

Returns new named pitch.

### 20.3.2 `pitchtools.clef_and_staff_position_number_to_named_pitch`

`pitchtools.clef_and_staff_position_number_to_named_pitch` (*clef*,  
*staff\_position\_number*)

Change *clef* and *staff\_position\_number* to named pitch:

```
>>> clef = contexttools.ClefMark('treble')
>>> for n in range(-6, 6):
...     pitch = pitchtools.clef_and_staff_position_number_to_named_pitch(clef, n)
...     print '%s\t%s\t%s' % (clef.clef_name, n, pitch)
treble   -6 c'
treble   -5 d'
treble   -4 e'
treble   -3 f'
treble   -2 g'
treble   -1 a'
treble    0 b'
treble    1 c''
treble    2 d''
treble    3 e''
treble    4 f''
treble    5 g''
```

Returns named pitch.

### 20.3.3 `pitchtools.contains_subsegment`

`pitchtools.contains_subsegment` (*pitch\_class\_numbers*, *pitch\_numbers*)  
True when *pitch\_numbers* contain *pitch\_class\_numbers* as subsegment:

```
>>> pcs = [2, 7, 10]
>>> pitches = [6, 9, 12, 13, 14, 19, 22, 27, 28, 29, 32, 35]
>>> pitchtools.contains_subsegment(pcs, pitches)
True
```

Returns boolean.

### 20.3.4 `pitchtools.get_named_pitch_from_pitch_carrier`

`pitchtools.get_named_pitch_from_pitch_carrier` (*pitch\_carrier*)  
Get named pitch from *pitch\_carrier*:

```
>>> pitch = pitchtools.NamedPitch('df', 5)
>>> pitch
NamedPitch("df'")
>>> pitchtools.get_named_pitch_from_pitch_carrier(pitch)
NamedPitch("df'")
```

```
>>> note = Note(('df', 5), (1, 4))
>>> note
Note("df''4")
>>> pitchtools.get_named_pitch_from_pitch_carrier(note)
NamedPitch("df'")
```

```
>>> note = Note(('df', 5), (1, 4))
>>> note.note_head
NoteHead("df'")
>>> pitchtools.get_named_pitch_from_pitch_carrier(note.note_head)
NamedPitch("df'")
```

```
>>> chord = Chord(['df', 5], (1, 4))
>>> chord
Chord("<df''>4")
>>> pitchtools.get_named_pitch_from_pitch_carrier(chord)
NamedPitch("df'")
```

```
>>> pitchtools.get_named_pitch_from_pitch_carrier(13)
NamedPitch("cs'")
```

Raise missing pitch error when *pitch\_carrier* carries no pitch.

Raise extra pitch error when *pitch\_carrier* carries more than one pitch.

Returns named pitch.

### 20.3.5 `pitchtools.get_numbered_pitch_class_from_pitch_carrier`

`pitchtools.get_numbered_pitch_class_from_pitch_carrier` (*pitch\_carrier*)

Get numbered pitch-class from *pitch\_carrier*:

```
>>> note = Note("cs'4")
>>> pitchtools.get_numbered_pitch_class_from_pitch_carrier(note)
NumberedPitchClass(1)
```

Raise missing pitch error on empty chords.

Raise extra pitch error on many-note chords.

Returns numbered pitch-class.

### 20.3.6 `pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list`

`pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list` (*notes*, *subrun\_indicators*)

Insert and transpose nested subruns in *pitch\_class\_number\_list* according to *subrun\_indicators*:

```
>>> notes = [Note(p, (1, 4)) for p in [0, 2, 7, 9, 5, 11, 4]]
>>> subrun_indicators = [(0, [2, 4]), (4, [3, 1])]
>>> pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list(
... notes, subrun_indicators)

>>> t = []
>>> for x in notes:
...     try:
...         t.append(x.written_pitch.pitch_number)
...     except AttributeError:
...         t.append([y.written_pitch.pitch_number for y in x])

>>> t
[0, [5, 7], 2, [4, 0, 6, 11], 7, 9, 5, [10, 6, 8], 11, [7], 4]
```

Set *subrun\_indicators* to a list of zero or more (*index*, *length\_list*) pairs.

For each (*index*, *length\_list*) pair in *subrun\_indicators* the function will read *index* mod `len(notes)` and insert a subrun of length `length_list[0]` immediately after `notes[index]`, a subrun of length `length_list[1]` immediately after `notes[index+1]`, and, in general, a subrun of length `length_list[i]` immediately after `notes[index+i]`, for  $i < \text{length}(\text{length\_list})$ .

New subruns are wrapped with lists. These wrapper lists are designed to allow inspection of the structural changes to *notes* immediately after the function returns. For this reason most calls to this function will be followed by `notes = sequencetools.flatten_sequence(notes)`:

```
>>> for note in notes: note
...
Note("c'4")
```

```
[Note("f'4"), Note("g'4")]
Note("d'4")
[Note("e'4"), Note("c'4"), Note("fs'4"), Note("b'4")]
Note("g'4")
Note("a'4")
Note("f'4")
[Note("bf'4"), Note("fs'4"), Note("af'4")]
Note("b'4")
[Note("g'4")]
Note("e'4")
```

This function is designed to work on a built-in Python list of notes. This function is **not** designed to work on Abjad voices, staves or other containers because the function currently implements no spanner-handling. That is, this function is designed to be used during precomposition when other, similar abstract pitch transforms may be common.

Returns list of integers and / or floats.

### 20.3.7 `pitchtools.instantiate_pitch_and_interval_test_collection`

`pitchtools.instantiate_pitch_and_interval_test_collection()`

Instantiate pitch and interval test collection:

```
>>> for x in pitchtools.instantiate_pitch_and_interval_test_collection(): x
...
NumberedInversionEquivalentIntervalClass(1)
NamedInversionEquivalentIntervalClass('+M2')
NumberedInterval(+1)
NumberedIntervalClass(+1)
NamedInterval('+M2')
NamedIntervalClass('+M2')
NamedPitch('c')
NamedPitchClass('c')
NumberedPitch(1)
NumberedPitchClass(1)
```

Use to test pitch and interval interface consistency.

Returns list.

### 20.3.8 `pitchtools.inventory_aggregate_subsets`

`pitchtools.inventory_aggregate_subsets()`

Inventory aggregate subsets:

```
>>> U_star = pitchtools.inventory_aggregate_subsets()
>>> len(U_star)
4096
>>> for pcset in U_star[:20]:
...     pcset
PitchClassSet([])
PitchClassSet([0])
PitchClassSet([1])
PitchClassSet([0, 1])
PitchClassSet([2])
PitchClassSet([0, 2])
PitchClassSet([1, 2])
PitchClassSet([0, 1, 2])
PitchClassSet([3])
PitchClassSet([0, 3])
PitchClassSet([1, 3])
PitchClassSet([0, 1, 3])
PitchClassSet([2, 3])
PitchClassSet([0, 2, 3])
PitchClassSet([1, 2, 3])
PitchClassSet([0, 1, 2, 3])
PitchClassSet([4])
```

```
PitchClassSet([0, 4])
PitchClassSet([1, 4])
PitchClassSet([0, 1, 4])
```

There are 4096 subsets of the aggregate.

This is  $U^*$  in [Morris 1987].

Returns list of numbered pitch-class sets.

### 20.3.9 `pitchtools.iterate_named_pitch_pairs_in_expr`

`pitchtools.iterate_named_pitch_pairs_in_expr(expr)`

Iterate left-to-right, top-to-bottom named pitch pairs in *expr*:

```
>>> score = Score([])
>>> notes = [Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"), Note("g'4")]
>>> score.append(Staff(notes))
>>> notes = [Note(x, (1, 4)) for x in [-12, -15, -17]]
>>> score.append(Staff(notes))
>>> contexttools.ClefMark('bass')(score[1])
ClefMark('bass')(Staff{3})
```

```
>>> for pair in pitchtools.iterate_named_pitch_pairs_in_expr(score):
...     pair
...
(NamedPitch("c'"), NamedPitch('c'))
(NamedPitch("c'"), NamedPitch("d'"))
(NamedPitch('c'), NamedPitch("d'"))
(NamedPitch("d'"), NamedPitch("e'"))
(NamedPitch("d'"), NamedPitch('a, '))
(NamedPitch('c'), NamedPitch("e'"))
(NamedPitch('c'), NamedPitch('a, '))
(NamedPitch("e'"), NamedPitch('a, '))
(NamedPitch("e'"), NamedPitch("f'"))
(NamedPitch('a, '), NamedPitch("f'"))
(NamedPitch("f'"), NamedPitch("g'"))
(NamedPitch("f'"), NamedPitch('g, '))
(NamedPitch('a, '), NamedPitch("g'"))
(NamedPitch('a, '), NamedPitch('g, '))
(NamedPitch("g'"), NamedPitch('g, '))
```

Chords are handled correctly.

```
>>> chord_1 = Chord([0, 2, 4], (1, 4))
>>> chord_2 = Chord([17, 19], (1, 4))
>>> staff = Staff([chord_1, chord_2])
```

```
>>> for pair in pitchtools.iterate_named_pitch_pairs_in_expr(staff):
...     print pair
...
(NamedPitch("c'"), NamedPitch("d'"))
(NamedPitch("c'"), NamedPitch("e'"))
(NamedPitch("d'"), NamedPitch("e'"))
(NamedPitch("c'"), NamedPitch("f'"))
(NamedPitch("c'"), NamedPitch("g'"))
(NamedPitch("d'"), NamedPitch("f'"))
(NamedPitch("d'"), NamedPitch("g'"))
(NamedPitch("e'"), NamedPitch("f'"))
(NamedPitch("e'"), NamedPitch("g'"))
(NamedPitch("f'"), NamedPitch("g'"))
```

Returns generator.

### 20.3.10 `pitchtools.list_named_pitches_in_expr`

`pitchtools.list_named_pitches_in_expr(expr)`

List named pitches in *expr*:



```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> beam_spanner = spannertools.BeamSpanner(staff[:])
```

```
>>> for x in pitchtools.list_named_pitches_in_expr(
...     beam_spanner):
...     x
...
NamedPitch("c'")
NamedPitch("d'")
NamedPitch("e'")
NamedPitch("f'")
```

Returns tuple.

### 20.3.11 pitchtools.list\_numbered\_interval\_numbers\_pairwise

`pitchtools.list_numbered_interval_numbers_pairwise` (*pitch\_carriers*, *wrap=False*)

List numbered interval numbers pairwise between *pitch\_carriers*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
```

```
>>> pitchtools.list_numbered_interval_numbers_pairwise(
... staff)
[2, 2, 1, 2, 2, 2, 1]
```

```
>>> pitchtools.list_numbered_interval_numbers_pairwise(
... staff, wrap=True)
[2, 2, 1, 2, 2, 2, 1, -12]
```

```
>>> notes = [
...     Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"),
...     Note("g'8"), Note("a'8"), Note("b'8"), Note("c''8")]
```

```
>>> notes.reverse()
```

```
>>> pitchtools.list_numbered_interval_numbers_pairwise(
... notes)
[-1, -2, -2, -2, -1, -2, -2]
```

```
>>> pitchtools.list_numbered_interval_numbers_pairwise(
... notes, wrap=True)
[-1, -2, -2, -2, -1, -2, -2, 12]
```

When `wrap = False` do not return `pitch_carriers[-1] - pitch_carriers[0]` as last in series.

When `wrap = True` do return `pitch_carriers[-1] - pitch_carriers[0]` as last in series.

Returns list.

### 20.3.12 pitchtools.list\_numbered\_inversion\_equivalent\_interval\_classes\_pairwise

`pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise` (*pitch\_carriers*, *wrap=False*)

List numbered inversion-equivalent interval-classes pairwise between *pitch\_carriers*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
```

```
>>> result = pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise(
... staff, wrap=False)
```

```
>>> for x in result: x
...
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
```

```
NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(1)
```

```
>>> result = pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise(
... staff, wrap=True)
```

```
>>> for x in result: x
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(0)
```

```
>>> notes = staff.select_leaves()
>>> notes = list(reversed(notes))
```

```
>>> result = pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise(
... notes, wrap=False)
```

```
>>> for x in result: x
...
NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
```

```
>>> result = pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise(
... notes, wrap=True)
```

```
>>> for x in result: x
...
NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(2)
NumberedInversionEquivalentIntervalClass(0)
```

When `wrap=False` do not return `pitch_carriers[-1]` - `pitch_carriers[0]` as last in series.

When `wrap=True` do return `pitch_carriers[-1]` - `pitch_carriers[0]` as last in series.

Returns list.

### 20.3.13 `pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range`

`pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range` (*pitch\_carrier*, *pitch\_range*)

List octave transpositions of *pitch\_carrier* in *pitch\_range*:

```
>>> chord = Chord("<c' d' e'>4")
>>> pitch_range = pitchtools.PitchRange(0, 48)
```

```
>>> result = pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range(
... chord, pitch_range)
```

```
>>> for chord in result:
...     chord
...
Chord("<c' d' e'>4")
Chord("<c'' d'' e''>4")
Chord("<c''' d''' e'''>4")
Chord("<c'''' d'''' e''''>4")
```

Returns list of newly created *pitch\_carrier* objects.

### 20.3.14 `pitchtools.list_ordered_named_pitch_pairs_from_expr_1_to_expr_2`

`pitchtools.list_ordered_named_pitch_pairs_from_expr_1_to_expr_2` (*expr\_1*,  
*expr\_2*)

List ordered named pitch pairs from *expr\_1* to *expr\_2*:

```
>>> chord_1 = Chord([0, 1, 2], (1, 4))
>>> chord_2 = Chord([3, 4], (1, 4))
```

```
>>> for pair in pitchtools.list_ordered_named_pitch_pairs_from_expr_1_to_expr_2(
...     chord_1, chord_2):
...     pair
(NamedPitch("c'"), NamedPitch("ef'"))
(NamedPitch("c'"), NamedPitch("e'"))
(NamedPitch("cs'"), NamedPitch("ef'"))
(NamedPitch("cs'"), NamedPitch("e'"))
(NamedPitch("d'"), NamedPitch("ef'"))
(NamedPitch("d'"), NamedPitch("e'"))
```

Returns generator.

### 20.3.15 `pitchtools.list_pitch_numbers_in_expr`

`pitchtools.list_pitch_numbers_in_expr` (*expr*)

List pitch numbers in *expr*:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> pitchtools.list_pitch_numbers_in_expr(tuplet)
(0, 2, 4)
```

Returns tuple of zero or more numbers.

### 20.3.16 `pitchtools.list_unordered_named_pitch_pairs_in_expr`

`pitchtools.list_unordered_named_pitch_pairs_in_expr` (*expr*)

List unordered named pitch pairs in *expr*:

```
>>> chord = Chord("<c' cs' d' ef'>4")
```

```
>>> for pair in pitchtools.list_unordered_named_pitch_pairs_in_expr(chord):
...     pair
...
(NamedPitch("c'"), NamedPitch("cs'"))
(NamedPitch("c'"), NamedPitch("d'"))
(NamedPitch("c'"), NamedPitch("ef'"))
(NamedPitch("cs'"), NamedPitch("d'"))
(NamedPitch("cs'"), NamedPitch("ef'"))
(NamedPitch("d'"), NamedPitch("ef'"))
```

Returns generator.

### 20.3.17 `pitchtools.make_n_middle_c_centered_pitches`

`pitchtools.make_n_middle_c_centered_pitches` (*n*)

Make *n* middle-c centered pitches, where  $0 < n$ :

```
>>> for p in pitchtools.make_n_middle_c_centered_pitches(5): p
NamedPitch('f')
NamedPitch('a')
NamedPitch("c'")
NamedPitch("e'")
NamedPitch("g'")
```

```
>>> for p in pitchtools.make_n_middle_c_centered_pitches(4): p
NamedPitch('g')
NamedPitch('b')
NamedPitch("d'")
NamedPitch("f'")
```

Returns list of zero or more named pitches.

### 20.3.18 `pitchtools.named_pitch_and_clef_to_staff_position_number`

`pitchtools.named_pitch_and_clef_to_staff_position_number` (*pitch*, *clef*)

Change named *pitch* and *clef* to staff position number:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> clef = contexttools.ClefMark('treble')
>>> for note in staff:
...     written_pitch = note.written_pitch
...     number = pitchtools.named_pitch_and_clef_to_staff_position_number(
...         written_pitch, clef)
...     print '%s\t%s' % (written_pitch, number)
c'      -6
d'      -5
e'      -4
f'      -3
g'      -2
a'      -1
b'       0
c''      1
```

Returns integer.

### 20.3.19 `pitchtools.numbered_inversion_equivalent_interval_class_dictionary`

`pitchtools.numbered_inversion_equivalent_interval_class_dictionary` (*pitches*)

Change named *pitches* to numbered inversion-equivalent interval-class number dictionary:

```
>>> chord = Chord("<c' d' b''>4")
>>> vector = pitchtools.numbered_inversion_equivalent_interval_class_dictionary(
...     chord.written_pitches)
>>> for i in range(7):
...     print '\t%s\t%s' % (i, vector[i])
...
0  0
1  1
2  1
3  1
4  0
5  0
6  0
```

Returns dictionary.

### 20.3.20 `pitchtools.permute_named_pitch_carrier_list_by_twelve_tone_row`

`pitchtools.permute_named_pitch_carrier_list_by_twelve_tone_row` (*pitches*, *row*)

Permute named pitch carrier list by twelve-tone *row*:

```
>>> notes = notetools.make_notes([17, -10, -2, 11], [Duration(1, 4)])
>>> row = pitchtools.TwelveToneRow([10, 0, 2, 6, 8, 7, 5, 3, 1, 9, 4, 11])
>>> pitchtools.permute_named_pitch_carrier_list_by_twelve_tone_row(notes, row)
[Note('bf4'), Note('d4'), Note('f'4'), Note('b'4")]
```

Function works by reference only. No objects are copied.

Returns list.

### 20.3.21 `pitchtools.register_pitch_class_numbers_by_pitch_number_aggregate`

`pitchtools.register_pitch_class_numbers_by_pitch_number_aggregate` (*pitch\_class\_numbers*, *aggregate*)

Register *pitch\_class\_numbers* by pitch-number *aggregate*:

```
>>> pitchtools.register_pitch_class_numbers_by_pitch_number_aggregate(
...     [10, 0, 2, 6, 8, 7, 5, 3, 1, 9, 4, 11],
...     [10, 19, 20, 23, 24, 26, 27, 29, 30, 33, 37, 40])
[10, 24, 26, 30, 20, 19, 29, 27, 37, 33, 40, 23]
```

Returns list of zero or more pitch numbers.

### 20.3.22 `pitchtools.set_written_pitch_of_pitched_components_in_expr`

`pitchtools.set_written_pitch_of_pitched_components_in_expr` (*expr*, *written\_pitch=0*)

Set written pitch of pitched components in *expr* to *written\_pitch*:

```
>>> staff = Staff("c' d' e' f'")
```

```
>>> pitchtools.set_written_pitch_of_pitched_components_in_expr(staff)
```

Use as a way of neutralizing pitch information in an arbitrary piece of score.

Returns none.

### 20.3.23 `pitchtools.sort_named_pitch_carriers_in_expr`

`pitchtools.sort_named_pitch_carriers_in_expr` (*pitch\_carriers*)

List named pitch carriers in *expr* sorted by numbered pitch-class:

```
>>> notes = notetools.make_notes([9, 11, 12, 14, 16], (1, 4))
```

```
>>> pitchtools.sort_named_pitch_carriers_in_expr(notes)
[Note("c'4"), Note("d'4"), Note("e'4"), Note("a'4"), Note("b'4")]
```

The elements in *pitch\_carriers* are not changed in any way.

Returns list.

### 20.3.24 `pitchtools.spell_numbered_interval_number`

`pitchtools.spell_numbered_interval_number` (*named\_interval\_number*, *numbered\_interval\_number*)

Spell *numbered\_interval\_number* according to *named\_interval\_number*:

```
>>> pitchtools.spell_numbered_interval_number(2, 1)
NamedInterval('+m2')
```

Returns named interval.

### 20.3.25 pitchtools.spell\_pitch\_number

`pitchtools.spell_pitch_number(pitch_number, diatonic_pitch_class_name)`  
Spell *pitch\_number* according to *diatonic\_pitch\_class\_name*:

```
>>> pitchtools.spell_pitch_number(14, 'c')
(Accidental('ss'), 5)
```

Returns accidental / octave-number pair.

### 20.3.26 pitchtools.suggest\_clef\_for\_named\_pitches

`pitchtools.suggest_clef_for_named_pitches(pitches)`  
Suggest clef for named *pitches*:

```
>>> staff = Staff(notetools.make_notes(range(-12, -6), [(1, 4)]))
>>> pitchtools.suggest_clef_for_named_pitches(staff)
ClefMark('bass')
```

Suggest clef based on minimal number of ledger lines.

Returns clef mark.

### 20.3.27 pitchtools.transpose\_named\_pitch\_by\_numbered\_interval\_and\_respell

`pitchtools.transpose_named_pitch_by_numbered_interval_and_respell(pitch, staff_spaces, num-bered_interval)`  
Transpose named pitch by *numbered\_interval* and respell *staff\_spaces* above or below:

```
>>> pitch = pitchtools.NamedPitch(0)

>>> pitchtools.transpose_named_pitch_by_numbered_interval_and_respell(
...     pitch, 1, 0.5)
NamedPitch("dtqf' ")
```

Returns new named pitch.

### 20.3.28 pitchtools.transpose\_pitch\_carrier\_by\_interval

`pitchtools.transpose_pitch_carrier_by_interval(pitch_carrier, interval)`  
Transpose *pitch\_carrier* by named *interval*:

```
>>> chord = Chord("<c' e' g'>4")

>>> pitchtools.transpose_pitch_carrier_by_interval(
...     chord, '+m2')
Chord("<df' f' af'>4")
```

Transpose *pitch\_carrier* by numbered *interval*:

```
>>> chord = Chord("<c' e' g'>4")
```

```
>>> pitchtools.transpose_pitch_carrier_by_interval(chord, 1)
Chord("<cs' f' af'>4")
```

Returns non-pitch-carrying input unchanged:

```
>>> rest = Rest('r4')
```

```
>>> pitchtools.transpose_pitch_carrier_by_interval(rest, 1)
Rest('r4')
```

Return *pitch\_carrier*.

### 20.3.29 pitchtools.transpose\_pitch\_class\_number\_to\_pitch\_number\_neighbor

`pitchtools.transpose_pitch_class_number_to_pitch_number_neighbor` (*pitch\_number*, *pitch\_class\_number*)

Transpose *pitch\_class\_number* by octaves to nearest neighbor of *pitch\_number*:

```
>>> pitchtools.transpose_pitch_class_number_to_pitch_number_neighbor(
...     12, 4)
16
```

Resulting pitch number must be within one tritone of *pitch\_number*.

Returns pitch number.

### 20.3.30 pitchtools.transpose\_pitch\_expr\_into\_pitch\_range

`pitchtools.transpose_pitch_expr_into_pitch_range` (*pitch\_expr*, *pitch\_range*)

Transpose *pitch\_expr* into *pitch\_range*:

```
>>> pitchtools.transpose_pitch_expr_into_pitch_range(
...     [-2, -1, 13, 14], pitchtools.PitchRange(0, 12))
[10, 11, 1, 2]
```

Returns new *pitch\_expr* object.

### 20.3.31 pitchtools.transpose\_pitch\_number\_by\_octave\_transposition\_mapping

`pitchtools.transpose_pitch_number_by_octave_transposition_mapping` (*pitch\_number*, *mapping*)

Transpose *pitch\_number* by the some number of octaves up or down. Derive correct number of octaves from *mapping* where *mapping* is a list of (*range\_spec*, *octave*) pairs and *range\_spec* is, in turn, a (*start*, *stop*) pair suitable to pass to the built-in Python `range()` function:

```
>>> mapping = [((-39, -13), 0), ((-12, 23), 12), ((24, 48), 24)]
```

The mapping given here comprises three (*range\_spec*, *octave*) pairs. The first such pair is `((-39, -13), 0)` and can be read as follows: “any pitches between -39 and -13 should be transposed into the octave rooted at pitch 0.” The octave rooted at pitch 0 equals the twelve pitches `range(0, 0 + 12)` or `[0, 1, ..., 10, 11]`.

The second (*range\_spec*, *octave*) pair is `((-12, 23), 12)` and can be read as “any pitches between -12 and 23 should be transposed into the octave rooted at pitch 12,” with the octave rooted at pitch 12 equal to the twelve pitches `range(12, 12 + 12)` or `[12, 13, ..., 22, 23]`.

The third and last (*range\_spec*, *octave*) pair is `((24, 48), 24)` and can be read as “any pitches between 24 and 48 should be transposed to the octave rooted at 24,” with the octave rooted at 24 equal to the twelve pitches `range(24, 24 + 12)` or `[24, 25, ..., 34, 35]`.

The mapping given here divides the compass of the piano, from  $-39$  to  $48$ , into three disjunct subranges and then explains how to transpose pitches found in any of those three disjunct subranges. This means that, for example, all the f-sharps within the range of the piano now undergo a known transposition under *mapping* as defined here:

```
>>> pitchtools.transpose_pitch_number_by_octave_transposition_mapping(  
...     -30, mapping)  
6
```

We verify that pitch  $-30$  should map to pitch  $6$  by noticing that pitch  $-30$  falls in the first of the three subranges defined by *mapping* from  $-39$  to  $-13$  and then noting that *mapping* sends pitches with that subrange to the octave rooted at pitch  $0$ . The octave transposition of  $-30$  that falls within the octave rooted at  $0$  is  $6$ :

```
>>> pitchtools.transpose_pitch_number_by_octave_transposition_mapping(  
...     -18, mapping)  
6
```

Likewise, *mapping* sends pitch  $-18$  to pitch  $6$  because pitch  $-18$  falls in the same subrange from  $-39$  to  $-13$  as did pitch  $-39$  and so undergoes the same transposition to the octave rooted at  $0$ .

In this way we can map all f-sharps from  $-39$  to  $48$  according to *mapping*:

```
>>> pitch_numbers = [-30, -18, -6, 6, 18, 30, 42]  
>>> for n in pitch_numbers:  
...     n, pitchtools.transpose_pitch_number_by_octave_transposition_mapping(  
...         n, mapping)  
...  
(-30, 6)  
(-18, 6)  
(-6, 18)  
(6, 18)  
(18, 18)  
(30, 30)  
(42, 30)
```

And so on.

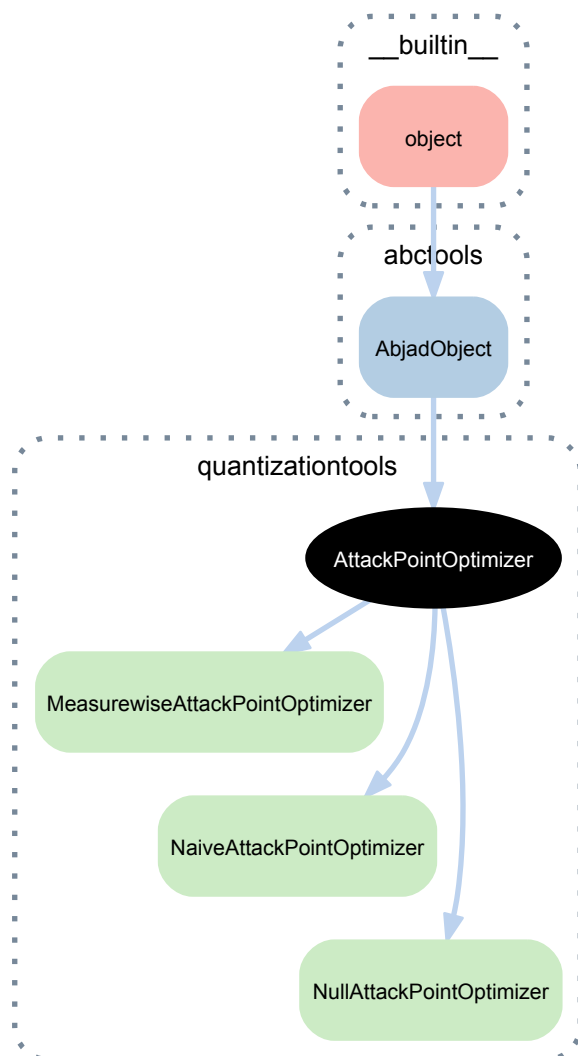
Returns pitch number.



## QUANTIZATIONTOOLS

### 21.1 Abstract classes

#### 21.1.1 quantizationtools.AttackPointOptimizer



**class** `quantizationtools.AttackPointOptimizer`

Abstract attack-point optimizer class from which concrete attack-point optimizer classes inherit.

Attack-point optimizers may alter the number, order, and individual durations of leaves in a tie chain, but may not alter the overall duration of that tie chain.

They effectively “clean up” notation, post-quantization.

## **Bases**

- `abctools.AbjadObject`
- `__builtin__.object`

## **Special methods**

`AttackPointOptimizer.__call__(expr)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

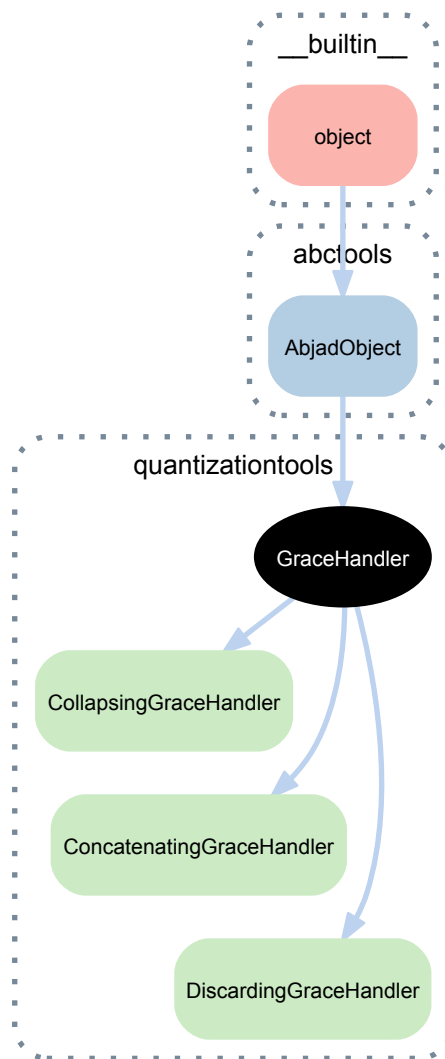
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 21.1.2 quantizationtools.GraceHandler



**class** `quantizationtools.GraceHandler`

Abstract base class from which concrete `GraceHandler` subclasses inherit.

Determines what pitch, if any, will be selected from a list of `QEvents` to be applied to an attack-point generated by a `QGrid`, and whether there should be a `GraceContainer` attached to th at attack-point.

When called on a sequence of `QEvents`, `GraceHandler` subclasses should return a pair, where the first item of the pair is a sequence of pitch tokens or `None`, and where the second item of the pair is a `GraceContainer` instance or `None`.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`GraceHandler.__call__(q_events)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

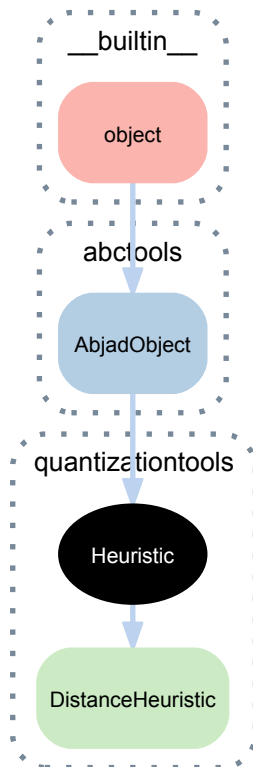
(AbjadObject).**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
Interpreter representation of Abjad object.

Returns string.

### 21.1.3 quantizationtools.Heuristic



**class** quantizationtools.**Heuristic**

Abstract base class from which concrete *Heuristic* subclasses inherit.

Heuristics rank *QGrids* according to the criteria they encapsulate.

They provide the means by which the quantizer selects a single *QGrid* from all computed *QGrids* for any given *QTargetBeat* to represent that beat.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

*Heuristic*.**\_\_call\_\_**(*q\_target\_beats*)

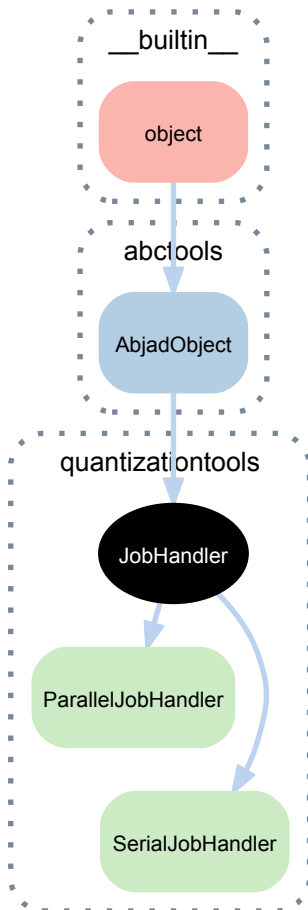
(AbjadObject).**\_\_eq\_\_**(*expr*)  
True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 21.1.4 quantizationtools.JobHandler



**class** quantizationtools.**JobHandler**

Abstract job handler class from which concrete job handlers inherit.

JobHandlers control how QuantizationJob instances are processed by the Quantizer, either serially or in parallel.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

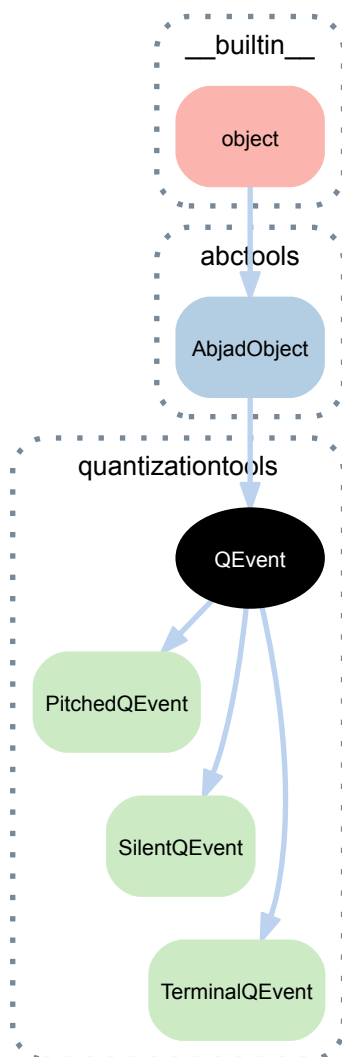
JobHandler.**\_\_call\_\_**(*jobs*)

(AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

## 21.1.5 quantizationtools.QEvent



**class** quantizationtools.**QEvent** (*offset, index=None*)  
 Abstract base class from which concrete `QEvent` subclasses inherit.  
 Represents an attack point to be quantized.  
 All `QEvents` possess a rational offset in milliseconds, and an optional index for disambiguating events which fall on the same offset in a `QGrid`.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `QEvent.index`

The optional index, for sorting QEvents with identical offsets.

### `QEvent.offset`

The offset in milliseconds of the event.

### `QEvent.storage_format`

Storage format of q-event.

Returns string.

## Special methods

### `(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

### `QEvent.__getstate__()`

### `QEvent.__lt__(expr)`

### `(AbjadObject).__ne__(expr)`

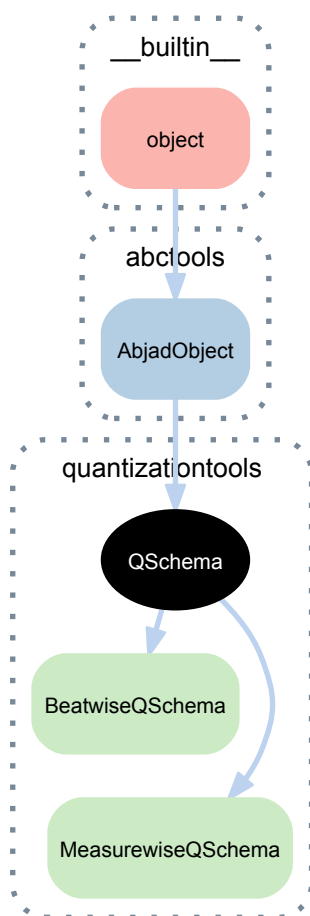
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

### `QEvent.__repr__()`

### `QEvent.__setstate__(state)`

### 21.1.6 quantizationtools.QSchema



**class** `quantizationtools.QSchema` (\*args, \*\*kwargs)

The *schema* for a quantization run.

`QSchema` allows for the specification of quantization settings diachronically, at any time-step of the quantization process.

In practice, this provides a means for the composer to change the tempo, search-tree, time-signature etc., effectively creating a template into which quantized rhythms can be “poured”, without yet knowing what those rhythms might be, or even how much time the ultimate result will take. Like Abjad’s `ContextMarks`, the settings made at any given time-step via a `QSchema` instance are understood to persist until changed.

All concrete `QSchema` subclasses strongly implement default values for all of their parameters.

*QSchema* is abstract.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`QSchema.item_class`

The schema’s item class.

`QSchema.items`

The item dictionary.



`QSchema.search_tree`

The default search tree.

`QSchema.target_class`

The schema's target class.

`QSchema.target_item_class`

The schema's target class' item class.

`QSchema.tempo`

The default tempo.

## Special methods

`QSchema.__call__(duration)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`QSchema.__getitem__(i)`

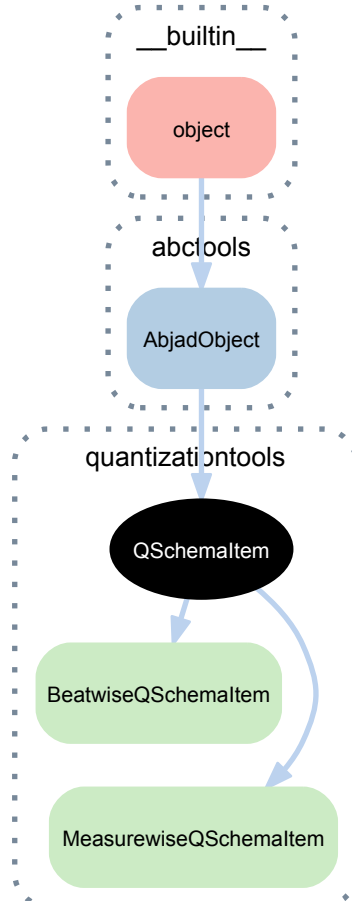
`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`QSchema.__repr__()`

## 21.1.7 quantizationtools.QSchemaltem



**class** `quantizationtools.QSchemaItem` (*search\_tree=None, tempo=None*)  
*QSchemaItem* represents a change of state in the timeline of a quantization process.  
*QSchemaItem* is abstract and immutable.

### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`QSchemaItem.search_tree`  
The optionally defined search tree.  
Returns search tree or none.

`QSchemaItem.storage_format`  
Storage format of q-schema item.  
Returns string.

`QSchemaItem.tempo`  
The optionally defined tempo mark.  
Returns tempo mark or none.

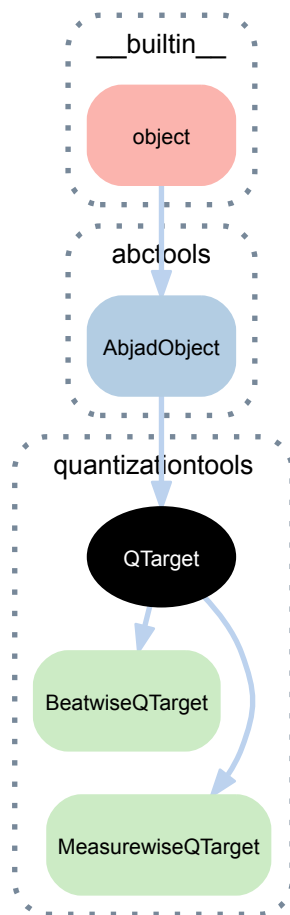
### Special methods

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

### 21.1.8 quantizationtools.QTarget



**class** `quantizationtools.QTarget` (*items*)

Abstract base class from which concrete `QTarget` subclasses inherit.

`QTarget` is created by a concrete `QSchema` instance, and represents the mold into which the timepoints contained by a `QSequence` instance will be poured, as structured by that `QSchema` instance.

Not composer-safe.

Used internally by the `Quantizer`.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`QTarget.beats`

`QTarget.duration_in_ms`

`QTarget.item_class`

`QTarget.items`

## Special methods

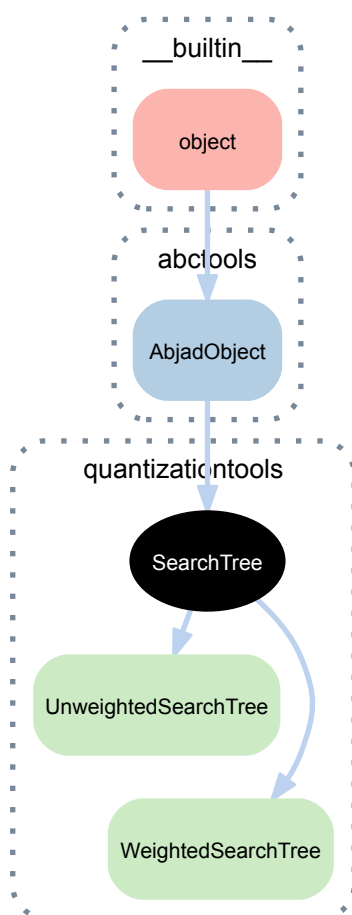
`QTarget.__call__(q_event_sequence, grace_handler=None, heuristic=None, job_handler=None, attack_point_optimizer=None, attach_tempo_marks=True)`

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 21.1.9 quantizationtools.SearchTree



**class** `quantizationtools.SearchTree` (*definition=None*)

Abstract base class from which concrete `SearchTree` subclasses inherit.

`SearchTrees` encapsulate strategies for generating collections of `QGrids`, given a set of `QEventProxy` instances as input.

They allow composers to define the degree and quality of nested rhythmic subdivisions in the quantization output. That is to say, they allow composers to specify what sorts of tuplets and ratios of pulses may be contained within other tuplets, to arbitrary levels of nesting.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`SearchTree.default_definition`

The default search tree definition.

Returns dictionary.

`SearchTree.definition`

The search tree definition.

Returns dictionary.

## Special methods

`SearchTree.__call__(q_grid)`

`SearchTree.__eq__(expr)`

`SearchTree.__getstate__()`

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

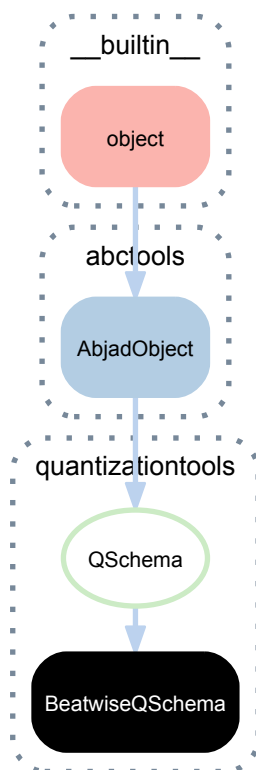
Returns boolean.

`SearchTree.__repr__()`

`SearchTree.__setstate__(state)`

## 21.2 Concrete classes

### 21.2.1 quantizationtools.BeatwiseQSchema



**class** `quantizationtools.BeatwiseQSchema` (*\*args, \*\*kwargs*)  
Concrete QSchema subclass which treats “beats” as its time-step unit:

```
>>> q_schema = quantizationtools.BeatwiseQSchema()
```

Without arguments, it uses smart defaults:

```
>>> q_schema
quantizationtools.BeatwiseQSchema(
  beatspan=durationtools.Duration(1, 4),
  search_tree=quantizationtools.UnweightedSearchTree(
    definition={ 2: { 2: { 2: { 2: None}, 3: None}, 3: None, 5: None, 7: None},
                 3: { 2: { 2: None}, 3: None, 5: None},
                 5: { 2: None, 3: None},
                 7: { 2: None},
                 11: None,
                 13: None}
  ),
  tempo=contexttools.TempoMark(
    durationtools.Duration(1, 4),
    60
  ),
)
```

Each time-step in a `BeatwiseQSchema` is composed of three settings:

- `beatspan`
- `search_tree`
- `tempo`

These settings can be applied as global defaults for the schema via keyword arguments, which persist until overridden:

```
>>> beatspan = Duration(5, 16)
>>> search_tree = quantizationtools.UnweightedSearchTree({7: None})
>>> tempo = contexttools.TempoMark((1, 4), 54)
>>> q_schema = quantizationtools.BeatwiseQSchema(
...     beatspan=beatspan,
...     search_tree=search_tree,
...     tempo=tempo,
... )
```

The computed value at any non-negative time-step can be found by subscripting:

```
>>> index = 0
>>> for key, value in sorted(q_schema[index].items()): print '{}:'.format(key), value
...
beatspan: 5/16
search_tree: UnweightedSearchTree(
    definition={ 7: None}
)
tempo: TempoMark(Duration(1, 4), 54)
```

```
>>> index = 1000
>>> for key, value in sorted(q_schema[index].items()): print '{}:'.format(key), value
...
beatspan: 5/16
search_tree: UnweightedSearchTree(
    definition={ 7: None}
)
tempo: TempoMark(Duration(1, 4), 54)
```

Per-time-step settings can be applied in a variety of ways.

Instantiating the schema via `*args` with a series of either `BeatwiseQSchemaItem` instances, or dictionaries which could be used to instantiate `BeatwiseQSchemaItem` instances, will apply those settings sequentially, starting from time-step 0:

```
>>> a = {'beatspan': Duration(5, 32)}
>>> b = {'beatspan': Duration(3, 16)}
>>> c = {'beatspan': Duration(1, 8)}
```

```
>>> q_schema = quantizationtools.BeatwiseQSchema(a, b, c)
```

```
>>> q_schema[0]['beatspan']
Duration(5, 32)
```

```
>>> q_schema[1]['beatspan']
Duration(3, 16)
```

```
>>> q_schema[2]['beatspan']
Duration(1, 8)
```

```
>>> q_schema[3]['beatspan']
Duration(1, 8)
```

Similarly, instantiating the schema from a single dictionary, consisting of integer:specification pairs, or a sequence via `*args` of (integer, specification) pairs, allows for applying settings to non-sequential time-steps:

```
>>> a = {'search_tree': quantizationtools.UnweightedSearchTree({2: None})}
>>> b = {'search_tree': quantizationtools.UnweightedSearchTree({3: None})}
```

```
>>> settings = {
...     2: a,
...     4: b,
... }
```

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(settings)
```

```
>>> q_schema[0]['search_tree']
UnweightedSearchTree(
  definition={ 2: { 2: { 2: { 2: None}, 3: None}, 3: None, 5: None, 7: None},
    3: { 2: { 2: None}, 3: None, 5: None},
    5: { 2: None, 3: None},
    7: { 2: None},
    11: None,
    13: None}
)
```

```
>>> q_schema[1]['search_tree']
UnweightedSearchTree(
  definition={ 2: { 2: { 2: { 2: None}, 3: None}, 3: None, 5: None, 7: None},
    3: { 2: { 2: None}, 3: None, 5: None},
    5: { 2: None, 3: None},
    7: { 2: None},
    11: None,
    13: None}
)
```

```
>>> q_schema[2]['search_tree']
UnweightedSearchTree(
  definition={ 2: None}
)
```

```
>>> q_schema[3]['search_tree']
UnweightedSearchTree(
  definition={ 2: None}
)
```

```
>>> q_schema[4]['search_tree']
UnweightedSearchTree(
  definition={ 3: None}
)
```

```
>>> q_schema[1000]['search_tree']
UnweightedSearchTree(
  definition={ 3: None}
)
```

The following is equivalent to the above schema definition:

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(
...     (2, {'search_tree': quantizationtools.UnweightedSearchTree({2: None}))),
...     (4, {'search_tree': quantizationtools.UnweightedSearchTree({3: None}))),
...     )
```

Return BeatwiseQSchema instance.

## Bases

- `quantizationtools.QSchema`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`BeatwiseQSchema.beatspan`  
The default beatspan.

`BeatwiseQSchema.item_class`  
The schema's item class.

`(QSchema).items`  
The item dictionary.



`(QSchema).search_tree`

The default search tree.

`BeatwiseQSchema.target_class`

`BeatwiseQSchema.target_item_class`

`(QSchema).tempo`

The default tempo.

## Special methods

`(QSchema).__call__(duration)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(QSchema).__getitem__(i)`

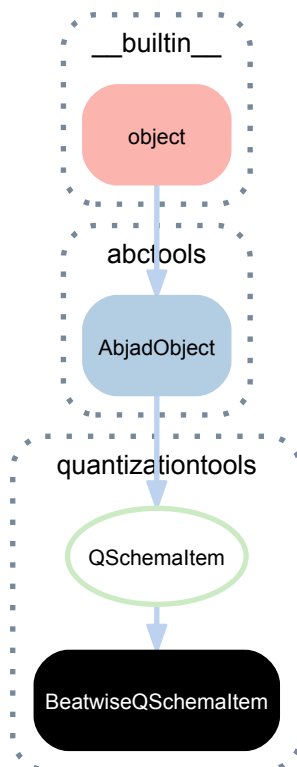
`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(QSchema).__repr__()`

### 21.2.2 quantizationtools.BeatwiseQSchemaItem



```
class quantizationtools.BeatwiseQSchemaItem(beatspan=None, search_tree=None,
                                             tempo=None)
```

*BeatwiseQSchemaItem* represents a change of state in the timeline of an unmetred quantization process.

```
>>> q_schema_item = quantizationtools.BeatwiseQSchemaItem()
>>> print q_schema_item.storage_format
quantizationtools.BeatwiseQSchemaItem()
```

Define a change in tempo:

```
>>> q_schema_item = quantizationtools.BeatwiseQSchemaItem(
...     tempo=((1, 4), 60),
...     )
>>> print q_schema_item.storage_format
quantizationtools.BeatwiseQSchemaItem(
    tempo=contexttools.TempoMark(
        durationtools.Duration(1, 4),
        60
    )
)
```

Define a change in beatspan:

```
>>> q_schema_item = quantizationtools.BeatwiseQSchemaItem(
...     beatspan=(1, 8),
...     )
>>> print q_schema_item.storage_format
quantizationtools.BeatwiseQSchemaItem(
    beatspan=durationtools.Duration(1, 8)
)
```

Returns beat-wise q-schema item.

## Bases

- `quantizationtools.QSchemaItem`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`BeatwiseQSchemaItem.beatspan`

The optionally defined beatspan duration.

Returns duration or none.

`(QSchemaItem).search_tree`

The optionally defined search tree.

Returns search tree or none.

`(QSchemaItem).storage_format`

Storage format of q-schema item.

Returns string.

`(QSchemaItem).tempo`

The optionally defined tempo mark.

Returns tempo mark or none.

## Special methods

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

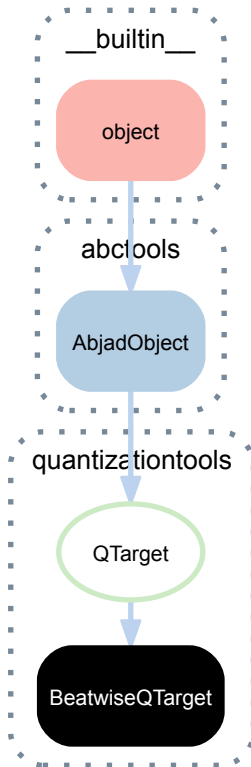
`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 21.2.3 quantizationtools.BeatwiseQTarget



**class** `quantizationtools.BeatwiseQTarget` (*items*)  
 A beat-wise quantization target.  
 Not composer-safe.  
 Used internally by `Quantizer`.  
 Return `BeatwiseQTarget` instance.

#### Bases

- `quantizationtools.QTarget`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`BeatwiseQTarget.beats`  
`(QTarget).duration_in_ms`  
`BeatwiseQTarget.item_class`  
`(QTarget).items`

## Special methods

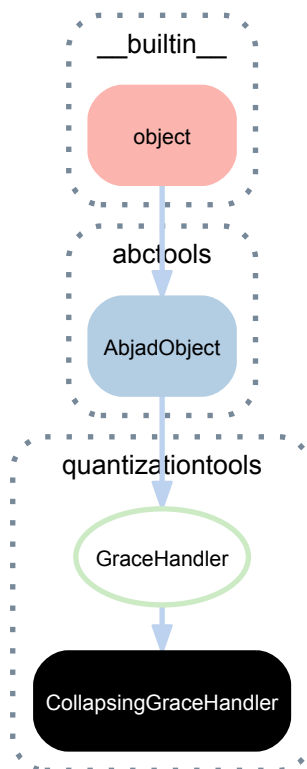
(QTarget) .**\_\_call\_\_**(*q\_event\_sequence*, *job\_handler=None*, *grace\_handler=None*, *heuristic=None*, *attack\_point\_optimizer=None*, *attach\_tempo\_marks=True*)

(AbjadObject) .**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject) .**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 21.2.4 quantizationtools.CollapsingGraceHandler



**class** quantizationtools.CollapsingGraceHandler

A GraceHandler which collapses pitch information into a single Chord, rather than creating a GraceContainer.

Return CollapsingGraceHandler instance.

## Bases

- quantizationtools.GraceHandler
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Special methods

`CollapsingGraceHandler.__call__(q_events)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

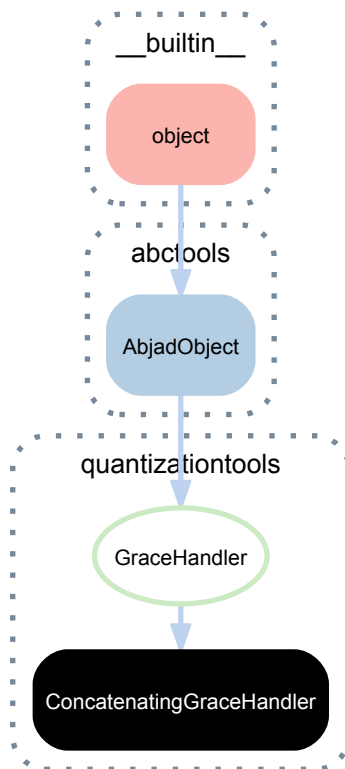
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 21.2.5 quantizationtools.ConcatenatingGraceHandler



**class** `quantizationtools.ConcatenatingGraceHandler` (*grace\_duration=None*)

Concrete `GraceHandler` subclass which concatenates all but the final `QEvent` attached to a `QGrid` offset into a `GraceContainer`, using a fixed leaf duration duration.

When called, it returns pitch information of final `QEvent`, and the generated `GraceContainer`, if any.

Return `ConcatenatingGraceHandler` instance.

## Bases

- `quantizationtools.GraceHandler`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ConcatenatingGraceHandler.grace_duration`

## Special methods

`ConcatenatingGraceHandler.__call__(q_events)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

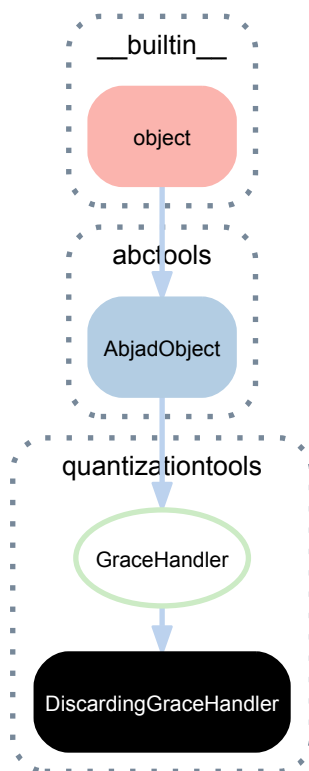
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 21.2.6 quantizationtools.DiscardingGraceHandler



**class** `quantizationtools.DiscardingGraceHandler`

Concrete `GraceHandler` subclass which discards all but final `QEvent` attached to an offset.

Does not create `GraceContainers`.

Return `DiscardingGraceHandler` instance.

## Bases

- `quantizationtools.GraceHandler`

- `abctools.AbjadObject`
- `__builtin__.object`

## Special methods

`DiscardingGraceHandler.__call__(q_events)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

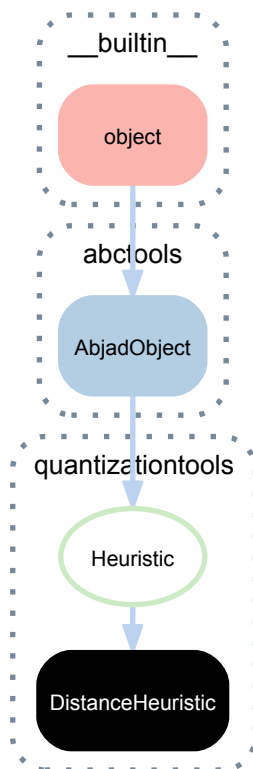
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 21.2.7 quantizationtools.DistanceHeuristic



**class** `quantizationtools.DistanceHeuristic`

Concrete `Heuristic` subclass which considers only the computed distance of each `QGrid` and the number of leaves of that `QGrid` when choosing the optimal `QGrid` for a given `QTargetBeat`.

The `QGrid` with the smallest distance and fewest number of leaves will be selected.

Return `DistanceHeuristic` instance.

## Bases

- `quantizationtools.Heuristic`

- `abctools.AbjadObject`
- `__builtin__.object`

### Special methods

(Heuristic) `.__call__(q_target_beats)`

(AbjadObject) `.__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject) `.__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

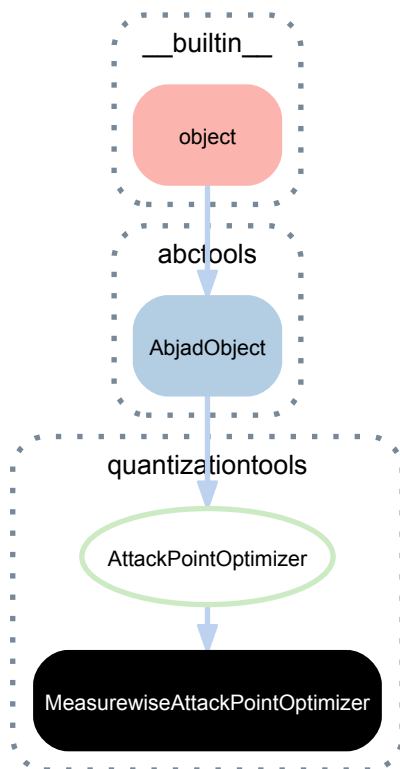
Returns boolean.

(AbjadObject) `.__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 21.2.8 quantizationtools.MeasurewiseAttackPointOptimizer



**class** `quantizationtools.MeasurewiseAttackPointOptimizer`

Concrete `AttackPointOptimizer` instance which attempts to optimize attack points in an expression with regard to the effective time signature of that expression.

Only acts on `Measure` instances.

Return `MeasurewiseAttackPointOptimizer` instance.

### Bases

- `quantizationtools.AttackPointOptimizer`



- `abctools.AbjadObject`
- `__builtin__.object`

### Special methods

`MeasurewiseAttackPointOptimizer.__call__(expr)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

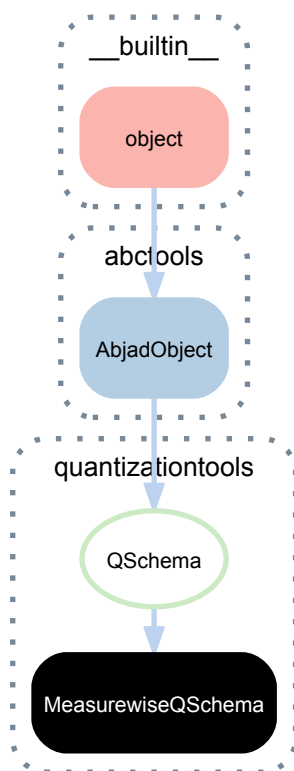
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 21.2.9 quantizationtools.MeasurewiseQSchema



**class** `quantizationtools.MeasurewiseQSchema` (*\*args, \*\*kwargs*)

Concrete QSchema subclass which treats “measures” as its time-step unit:

```
>>> q_schema = quantizationtools.MeasurewiseQSchema()
```

Without arguments, it uses smart defaults:

```
>>> q_schema
quantizationtools.MeasurewiseQSchema(
  search_tree=quantizationtools.UnweightedSearchTree(
    definition={ 2: { 2: { 2: { 2: None}, 3: None}, 3: None, 5: None, 7: None},
                 3: { 2: { 2: None}, 3: None, 5: None},
```

```
        5: { 2: None, 3: None},
        7: { 2: None},
        11: None,
        13: None}
    ),
    tempo=contexttools.TempoMark(
        durationtools.Duration(1, 4),
        60
    ),
    time_signature=contexttools.TimeSignatureMark(
        (4, 4)
    ),
    use_full_measure=False,
)
```

Each time-step in a `MeasurewiseQSchema` is composed of four settings:

- `search_tree`
- `tempo`
- `time_signature`
- `use_full_measure`

These settings can be applied as global defaults for the schema via keyword arguments, which persist until overridden:

```
>>> search_tree = quantizationtools.UnweightedSearchTree({7: None})
>>> time_signature = contexttools.TimeSignatureMark((3, 4))
>>> tempo = contexttools.TempoMark((1, 4), 54)
>>> use_full_measure = True
>>> q_schema = quantizationtools.MeasurewiseQSchema(
...     search_tree=search_tree,
...     tempo=tempo,
...     time_signature=time_signature,
...     use_full_measure=use_full_measure,
... )
```

All of these settings are self-descriptive, except for `use_full_measure`, which controls whether the measure is subdivided by the `Quantizer` into beats according to its time signature.

If `use_full_measure` is `False`, the time-step's measure will be divided into units according to its time-signature. For example, a 4/4 measure will be divided into 4 units, each having a beatspan of 1/4.

On the other hand, if `use_full_measure` is set to `True`, the time-step's measure will not be subdivided into independent quantization units. This usually results in full-measure tuplets.

The computed value at any non-negative time-step can be found by subscripting:

```
>>> index = 0
>>> for key, value in sorted(q_schema[index].items()): print '{}:'.format(key), value
...
search_tree: UnweightedSearchTree(
    definition={ 7: None}
)
tempo: TempoMark(Duration(1, 4), 54)
time_signature: 3/4
use_full_measure: True
```

```
>>> index = 1000
>>> for key, value in sorted(q_schema[index].items()): print '{}:'.format(key), value
...
search_tree: UnweightedSearchTree(
    definition={ 7: None}
)
tempo: TempoMark(Duration(1, 4), 54)
time_signature: 3/4
use_full_measure: True
```

Per-time-step settings can be applied in a variety of ways.

Instantiating the schema via `*args` with a series of either `MeasurewiseQSchemaItem` instances, or dictionaries which could be used to instantiate `MeasurewiseQSchemaItem` instances, will apply those settings sequentially, starting from time-step 0:

```
>>> a = {'search_tree': quantizationtools.UnweightedSearchTree({2: None})}
>>> b = {'search_tree': quantizationtools.UnweightedSearchTree({3: None})}
>>> c = {'search_tree': quantizationtools.UnweightedSearchTree({5: None})}
```

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(a, b, c)
```

```
>>> q_schema[0]['search_tree']
UnweightedSearchTree(
  definition={ 2: None}
)
```

```
>>> q_schema[1]['search_tree']
UnweightedSearchTree(
  definition={ 3: None}
)
```

```
>>> q_schema[2]['search_tree']
UnweightedSearchTree(
  definition={ 5: None}
)
```

```
>>> q_schema[1000]['search_tree']
UnweightedSearchTree(
  definition={ 5: None}
)
```

Similarly, instantiating the schema from a single dictionary, consisting of integer:specification pairs, or a sequence via `*args` of (integer, specification) pairs, allows for applying settings to non-sequential time-steps:

```
>>> a = {'time_signature': contexttools.TimeSignatureMark((7, 32))}
>>> b = {'time_signature': contexttools.TimeSignatureMark((3, 4))}
>>> c = {'time_signature': contexttools.TimeSignatureMark((5, 8))}
```

```
>>> settings = {
...     2: a,
...     4: b,
...     6: c,
... }
```

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(settings)
```

```
>>> q_schema[0]['time_signature']
TimeSignatureMark((4, 4))
```

```
>>> q_schema[1]['time_signature']
TimeSignatureMark((4, 4))
```

```
>>> q_schema[2]['time_signature']
TimeSignatureMark((7, 32))
```

```
>>> q_schema[3]['time_signature']
TimeSignatureMark((7, 32))
```

```
>>> q_schema[4]['time_signature']
TimeSignatureMark((3, 4))
```

```
>>> q_schema[5]['time_signature']
TimeSignatureMark((3, 4))
```

```
>>> q_schema[6]['time_signature']
TimeSignatureMark((5, 8))
```

```
>>> q_schema[1000]['time_signature']
TimeSignatureMark((5, 8))
```

The following is equivalent to the above schema definition:

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(
...     (2, {'time_signature': contexttools.TimeSignatureMark((7, 32))}),
...     (4, {'time_signature': contexttools.TimeSignatureMark((3, 4))}),
...     (6, {'time_signature': contexttools.TimeSignatureMark((5, 8))}),
...     )
```

Return `MeasurewiseQSchema` instance.

## Bases

- `quantizationtools.QSchema`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`MeasurewiseQSchema.item_class`

The schema's item class.

`(QSchema).items`

The item dictionary.

`(QSchema).search_tree`

The default search tree.

`MeasurewiseQSchema.target_class`

`MeasurewiseQSchema.target_item_class`

`(QSchema).tempo`

The default tempo.

`MeasurewiseQSchema.time_signature`

The default time signature.

`MeasurewiseQSchema.use_full_measure`

The full-measure-as-beatspan default.

## Special methods

`(QSchema).__call__(duration)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(QSchema).__getitem__(i)`

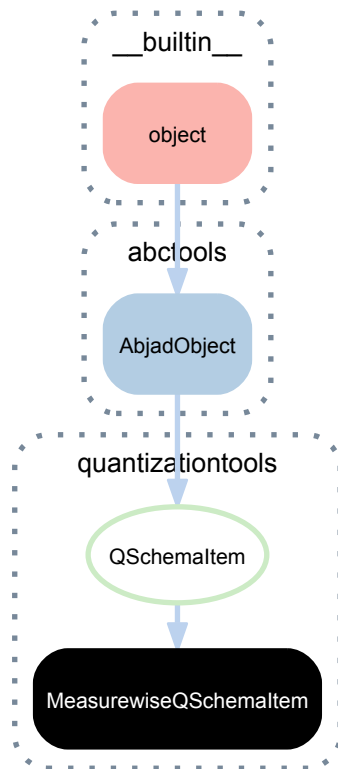
`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(QSchema).__repr__()`

### 21.2.10 quantizationtools.MeasurewiseQSchemaItem



**class** `quantizationtools.MeasurewiseQSchemaItem` (*search\_tree=None*, *tempo=None*,  
*time\_signature=None*,  
*use\_full\_measure=None*)

*MeasurewiseQSchemaItem* represents a change of state in the timeline of a metered quantization process.

```
>>> q_schema_item = quantizationtools.MeasurewiseQSchemaItem()
>>> print q_schema_item.storage_format
quantizationtools.MeasurewiseQSchemaItem()
```

Define a change in tempo:

```
>>> q_schema_item = quantizationtools.MeasurewiseQSchemaItem(
...     tempo=((1, 4), 60),
... )
>>> print q_schema_item.storage_format
quantizationtools.MeasurewiseQSchemaItem(
    tempo=contexttools.TempoMark(
        durationtools.Duration(1, 4),
        60
    )
)
```

Define a change in time signature:

```
>>> q_schema_item = quantizationtools.MeasurewiseQSchemaItem(
...     time_signature=(6, 8),
... )
>>> print q_schema_item.storage_format
quantizationtools.MeasurewiseQSchemaItem(
    time_signature=contexttools.TimeSignatureMark(
        (6, 8)
    )
)
```

Test for beatspan, given a defined time signature:

```
>>> q_schema_item.beatspan
Duration(1, 8)
```

*MeasurewiseQSchemaItem* is immutable.

Return *MeasurewiseQSchemaItem* instance.

## Bases

- `quantizationtools.QSchemaItem`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`MeasurewiseQSchemaItem.beatspan`

The beatspan duration, if a time signature was defined.

Returns duration or none.

`(QSchemaItem).search_tree`

The optionally defined search tree.

Returns search tree or none.

`(QSchemaItem).storage_format`

Storage format of q-schema item.

Returns string.

`(QSchemaItem).tempo`

The optionally defined tempo mark.

Returns tempo mark or none.

`MeasurewiseQSchemaItem.time_signature`

The optionally defined TimeSignatureMark.

Returns time signature mark or none

`MeasurewiseQSchemaItem.use_full_measure`

If True, use the full measure as the beatspan.

Returns boolean or none.

## Special methods

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

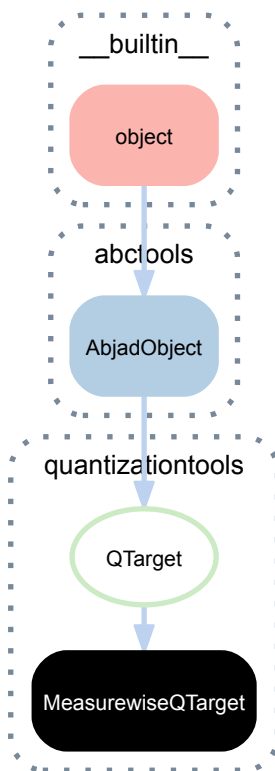
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 21.2.11 quantizationtools.MeasurewiseQTarget



**class** `quantizationtools.MeasurewiseQTarget` (*items*)

A measure-wise quantization target.

Not composer-safe.

Used internally by `Quantizer`.

#### Bases

- `quantizationtools.QTarget`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`MeasurewiseQTarget.beats`

`(QTarget).duration_in_ms`

`MeasurewiseQTarget.item_class`

`(QTarget).items`

#### Special methods

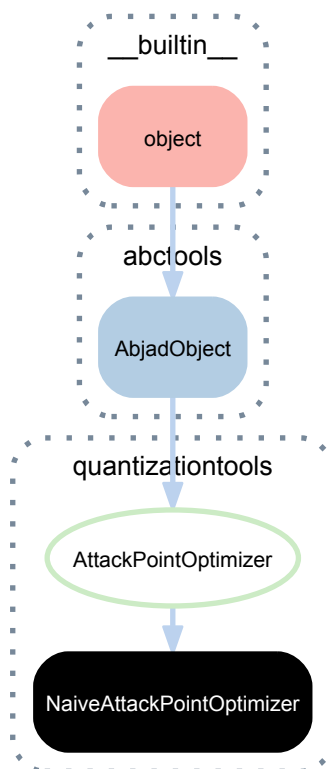
`(QTarget).__call__(q_event_sequence,` *grace\_handler=None,* *heuristic=None,*  
*job\_handler=None,* *attack\_point\_optimizer=None,* *at-*  
*tach\_tempo\_marks=True)*

(AbjadObject) .**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject) .**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

## 21.2.12 quantizationtools.NaiveAttackPointOptimizer



**class** quantizationtools.**NaiveAttackPointOptimizer**  
 Concrete AttackPointOptimizer subclass which optimizes attack points by fusing tie leaves within tie chains with leaf durations decreasing monotonically.  
 TieChains will be partitioned into sub-TieChains if leaves are found with TempoMarks attached.  
 Return NaiveAttackPointOptimizer instance.

### Bases

- quantizationtools.AttackPointOptimizer
- abctools.AbjadObject
- \_\_builtin\_\_.object

### Special methods

NaiveAttackPointOptimizer.**\_\_call\_\_**(*expr*)

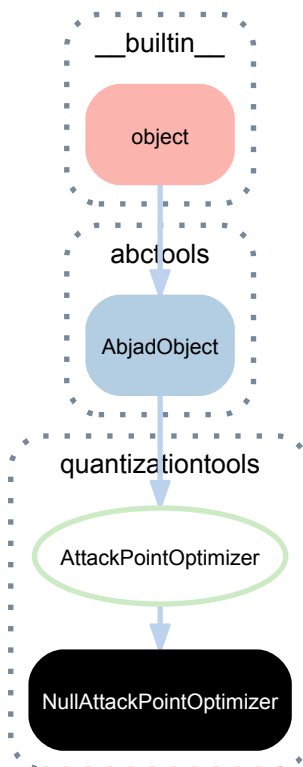


`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 21.2.13 quantizationtools.NullAttackPointOptimizer



**class** `quantizationtools.NullAttackPointOptimizer`  
 Concrete `AttackPointOptimizer` subclass which performs no attack point optimization.  
 Return `NullAttackPointOptimizer` instance.

#### Bases

- `quantizationtools.AttackPointOptimizer`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

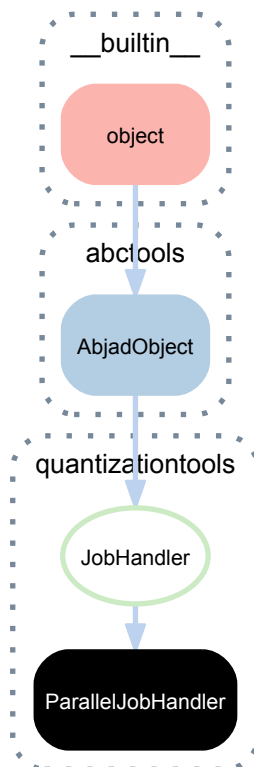
`NullAttackPointOptimizer.__call__(expr)`

(AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

## 21.2.14 quantizationtools.ParallelJobHandler



**class** quantizationtools.**ParallelJobHandler**  
 Processes QuantizationJob instances in parallel, based on the number of CPUs available.

### Bases

- quantizationtools.JobHandler
- abctools.AbjadObject
- \_\_builtin\_\_.object

### Special methods

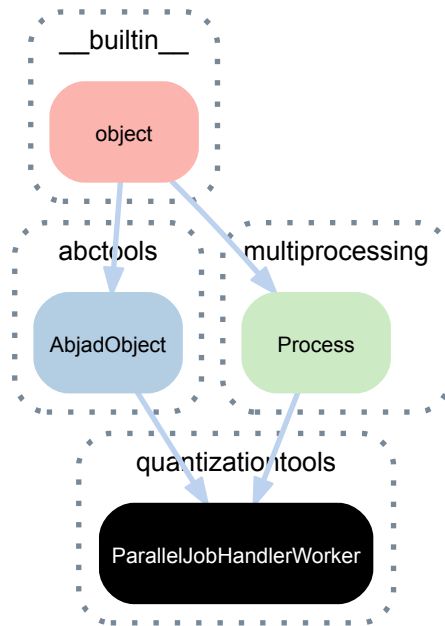
ParallelJobHandler.**\_\_call\_\_**(*jobs*)

(AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 21.2.15 quantizationtools.ParallelJobHandlerWorker



**class** quantizationtools.**ParallelJobHandlerWorker** (*job\_queue*, *result\_queue*)  
 Worker process which runs QuantizationJobs.  
 Not composer-safe.  
 Used internally by ParallelJobHandler.  
 Return ParallelJobHandlerWorker instance.

#### Bases

- multiprocessing.process.Process
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

(Process).**exitcode**  
 Return exit code of process or *None* if it has yet to stop

(Process).**ident**  
 Return identifier (PID) of process or *None* if it has yet to start

(Process).**pid**  
 Return identifier (PID) of process or *None* if it has yet to start

## Read/write properties

(Process) **.authkey**

(Process) **.daemon**

Return whether process is a daemon

(Process) **.name**

## Methods

(Process) **.is\_alive()**

Return whether process is alive

(Process) **.join** (*timeout=None*)

Wait until child process terminates

ParallelJobHandlerWorker **.run()**

(Process) **.start()**

Start child process

(Process) **.terminate()**

Terminate process; sends SIGTERM signal or uses TerminateProcess()

## Special methods

(AbjadObject) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

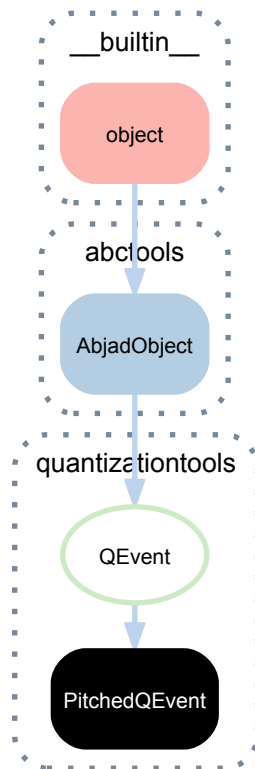
(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Process) **.\_\_repr\_\_** ()

### 21.2.16 quantizationtools.PitchedQEvent



**class** `quantizationtools.PitchedQEvent` (*offset, pitches, attachments=None, index=None*)  
 A QEvent which indicates the onset of a period of pitched material in a QEventSequence:

```

>>> pitches = [0, 1, 4]
>>> q_event = quantizationtools.PitchedQEvent(1000, pitches)
>>> q_event
quantizationtools.PitchedQEvent(
  durationtools.Offset(1000, 1),
  (NamedPitch("c'"), NamedPitch("cs'"), NamedPitch("e'")),
  attachments=()
)
  
```

#### Bases

- `quantizationtools.QEvent`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`PitchedQEvent.attachments`

`(QEvent).index`

The optional index, for sorting QEvents with identical offsets.

`(QEvent).offset`

The offset in milliseconds of the event.

`PitchedQEvent.pitches`

`PitchedQEvent.storage_format`

Storage format of pitch q-event.

Returns string.

## Special methods

`PitchedQEvent.__eq__(expr)`

`(QEvent).__getstate__()`

`(QEvent).__lt__(expr)`

`(AbjadObject).__ne__(expr)`

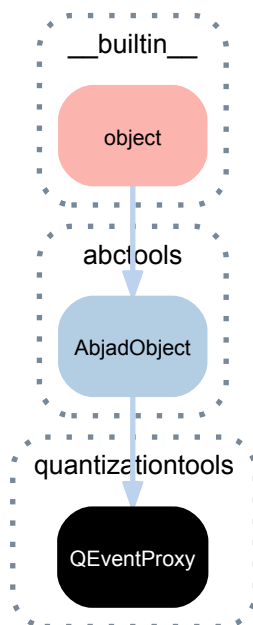
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(QEvent).__repr__()`

`(QEvent).__setstate__(state)`

### 21.2.17 quantizationtools.QEventProxy



**class** `quantizationtools.QEventProxy(*args)`

Proxies a *QEvent*, mapping that *QEvent*'s offset with the range of its beatspan to the range 0-1:

```

>>> q_event = quantizationtools.PitchedQEvent(130, [0, 1, 4])
>>> proxy = quantizationtools.QEventProxy(q_event, 0.5)
>>> proxy
quantizationtools.QEventProxy(
  quantizationtools.PitchedQEvent(
    durationtools.Offset(130, 1),
    (NamedPitch("c'"), NamedPitch("cs'"), NamedPitch("e'")),
    attachments=()
  ),
  durationtools.Offset(1, 2)
)
  
```

Not composer-safe.

Used internally by *Quantizer*.

Returns *QEventProxy* instance.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`QEventProxy.index`

`QEventProxy.offset`

`QEventProxy.q_event`

## Special methods

`QEventProxy.__eq__(expr)`

`QEventProxy.__getstate__()`

`(AbjadObject).__ne__(expr)`

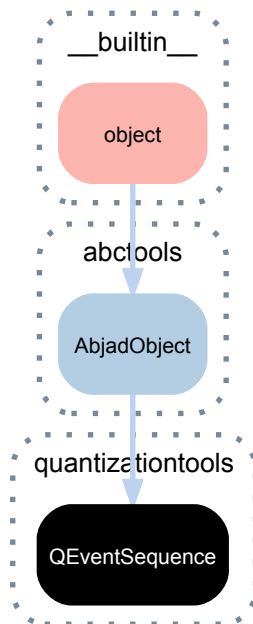
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`QEventProxy.__repr__()`

`QEventProxy.__setstate__(state)`

## 21.2.18 quantizationtools.QEventSequence



**class** `quantizationtools.QEventSequence` (*sequence*)

A well-formed sequence of q-events.

Contains only pitched q-events and silent q-events, and terminates with a single terminal q-event.

A q-event sequence is the primary input to the quantizer.

A q-event sequence provides a number of convenience functions to assist with instantiating new sequences:

```
>>> durations = (1000, -500, 1250, -500, 750)
```

```
>>> sequence = \
...     quantizationtools.QEventSequence.from_millisecond_durations(
...     durations)
```

```
>>> for q_event in sequence:
...     q_event
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedPitch("c'"),),
    attachments=()
)
quantizationtools.SilentQEvent(
    durationtools.Offset(1000, 1),
    attachments=()
)
quantizationtools.PitchedQEvent(
    durationtools.Offset(1500, 1),
    (NamedPitch("c'"),),
    attachments=()
)
quantizationtools.SilentQEvent(
    durationtools.Offset(2750, 1),
    attachments=()
)
quantizationtools.PitchedQEvent(
    durationtools.Offset(3250, 1),
    (NamedPitch("c'"),),
    attachments=()
)
quantizationtools.TerminalQEvent(
    durationtools.Offset(4000, 1)
)
```

Returns q-event sequence.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`QEventSequence.duration_in_ms`

The total duration in milliseconds of the `QEventSequence`:

```
>>> sequence.duration_in_ms
Duration(4000, 1)
```

Return `Duration` instance.

`QEventSequence.sequence`

The sequence of q-events:

```
>>> for q_event in sequence.sequence:
...     print q_event.storage_format
...
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedPitch("c'"),),
    attachments=()
)
quantizationtools.SilentQEvent(
    durationtools.Offset(1000, 1),
    attachments=()
)
```



```

    )
    quantizationtools.PitchedQEvent(
        durationtools.Offset(1500, 1),
        (NamedPitch("c'"),),
        attachments=()
    )
    quantizationtools.SilentQEvent(
        durationtools.Offset(2750, 1),
        attachments=()
    )
    quantizationtools.PitchedQEvent(
        durationtools.Offset(3250, 1),
        (NamedPitch("c'"),),
        attachments=()
    )
    quantizationtools.TerminalQEvent(
        durationtools.Offset(4000, 1)
    )

```

Returns tuple.

`QEventSequence.storage_format`

Storage format of Q event sequence.

```

>>> print sequence.storage_format
quantizationtools.QEventSequence(
  (quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedPitch("c'"),),
    attachments=()
  ), quantizationtools.SilentQEvent(
    durationtools.Offset(1000, 1),
    attachments=()
  ), quantizationtools.PitchedQEvent(
    durationtools.Offset(1500, 1),
    (NamedPitch("c'"),),
    attachments=()
  ), quantizationtools.SilentQEvent(
    durationtools.Offset(2750, 1),
    attachments=()
  ), quantizationtools.PitchedQEvent(
    durationtools.Offset(3250, 1),
    (NamedPitch("c'"),),
    attachments=()
  ), quantizationtools.TerminalQEvent(
    durationtools.Offset(4000, 1)
  ))
)

```

Returns string.

## Class methods

`QEventSequence.from_millisecond_durations` (*milliseconds*, *fuse\_silences=False*)

Convert a sequence of millisecond durations durations into a QEventSequence:

```

>>> durations = [-250, 500, -1000, 1250, -1000]

```

```

>>> sequence = \
...     quantizationtools.QEventSequence.from_millisecond_durations(
...         durations)

```

```

>>> for q_event in sequence:
...     q_event
...
quantizationtools.SilentQEvent(
    durationtools.Offset(0, 1),
    attachments=()
)

```

```
quantizationtools.PitchedQEvent(  
    durationtools.Offset(250, 1),  
    (NamedPitch("c'"),),  
    attachments=()  
)  
quantizationtools.SilentQEvent(  
    durationtools.Offset(750, 1),  
    attachments=()  
)  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(1750, 1),  
    (NamedPitch("c'"),),  
    attachments=()  
)  
quantizationtools.SilentQEvent(  
    durationtools.Offset(3000, 1),  
    attachments=()  
)  
quantizationtools.TerminalQEvent(  
    durationtools.Offset(4000, 1)  
)
```

Return QEventSequence instance.

QEventSequence.**from\_millisecond\_offsets** (*offsets*)

Convert millisecond offsets *offsets* into a QEventSequence:

```
>>> offsets = [0, 250, 750, 1750, 3000, 4000]
```

```
>>> sequence = \  
...     quantizationtools.QEventSequence.from_millisecond_offsets(  
...         offsets)
```

```
>>> for q_event in sequence:  
...     q_event  
...  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(0, 1),  
    (NamedPitch("c'"),),  
    attachments=()  
)  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(250, 1),  
    (NamedPitch("c'"),),  
    attachments=()  
)  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(750, 1),  
    (NamedPitch("c'"),),  
    attachments=()  
)  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(1750, 1),  
    (NamedPitch("c'"),),  
    attachments=()  
)  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(3000, 1),  
    (NamedPitch("c'"),),  
    attachments=()  
)  
quantizationtools.TerminalQEvent(  
    durationtools.Offset(4000, 1)  
)
```

Return QEventSequence instance.

QEventSequence.**from\_millisecond\_pitch\_pairs** (*pairs*)

Convert millisecond-duration:pitch pairs *pairs* into a QEventSequence:

```
>>> durations = [250, 500, 1000, 1250, 1000]
>>> pitches = [(0,), None, (2, 3), None, (1,)]
>>> pairs = zip(durations, pitches)
```

```
>>> sequence = \
...     quantizationtools.QEventSequence.from_millisecond_pitch_pairs(
...     pairs)
```

```
>>> for q_event in sequence:
...     q_event
...
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedPitch("c'"),),
    attachments=()
)
quantizationtools.SilentQEvent(
    durationtools.Offset(250, 1),
    attachments=()
)
quantizationtools.PitchedQEvent(
    durationtools.Offset(750, 1),
    (NamedPitch("d'"), NamedPitch("ef'")),
    attachments=()
)
quantizationtools.SilentQEvent(
    durationtools.Offset(1750, 1),
    attachments=()
)
quantizationtools.PitchedQEvent(
    durationtools.Offset(3000, 1),
    (NamedPitch("cs'"),),
    attachments=()
)
quantizationtools.TerminalQEvent(
    durationtools.Offset(4000, 1)
)
```

Return QEventSequence instance.

QEventSequence.**from\_tempo\_scaled\_durations** (*durations*, *tempo=None*)

Convert durations, scaled by tempo into a QEventSequence:

```
>>> tempo = contexttools.TempoMark((1, 4), 174)
>>> durations = [(1, 4), (-3, 16), (1, 16), (-1, 2)]
```

```
>>> sequence = \
...     quantizationtools.QEventSequence.from_tempo_scaled_durations(
...     durations, tempo=tempo)
```

```
>>> for q_event in sequence:
...     q_event
...
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedPitch("c'"),),
    attachments=()
)
quantizationtools.SilentQEvent(
    durationtools.Offset(10000, 29),
    attachments=()
)
quantizationtools.PitchedQEvent(
    durationtools.Offset(17500, 29),
    (NamedPitch("c'"),),
    attachments=()
)
quantizationtools.SilentQEvent(
    durationtools.Offset(20000, 29),
    attachments=()
)
```

```
quantizationtools.TerminalQEvent(  
    durationtools.Offset(40000, 29)  
)
```

Return `QEventSequence` instance.

`QEventSequence.from_tempo_scaled_leaves` (*leaves*, *tempo=None*)

Convert *leaves*, optionally with *tempo* into a `QEventSequence`:

```
>>> staff = Staff("c'4 <d' fs'>8. r16 ggs'2")  
>>> tempo = contexttools.TempoMark((1, 4), 72)
```

```
>>> sequence = \  
...     quantizationtools.QEventSequence.from_tempo_scaled_leaves(  
...         staff.select_leaves(), tempo=tempo)
```

```
>>> for q_event in sequence:  
...     q_event  
...  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(0, 1),  
    (NamedPitch("c'"),),  
    attachments=()  
)  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(2500, 3),  
    (NamedPitch("d'"), NamedPitch("fs'")),  
    attachments=()  
)  
quantizationtools.SilentQEvent(  
    durationtools.Offset(4375, 3),  
    attachments=()  
)  
quantizationtools.PitchedQEvent(  
    durationtools.Offset(5000, 3),  
    (NamedPitch("ggs'"),),  
    attachments=()  
)  
quantizationtools.TerminalQEvent(  
    durationtools.Offset(10000, 3)  
)
```

If *tempo* is `None`, all leaves in *leaves* must have an effective, non-imprecise tempo. The millisecond-duration of each leaf will be determined by its effective tempo.

Return `QEventSequence` instance.

## Special methods

`QEventSequence.__contains__` (*expr*)

`QEventSequence.__eq__` (*expr*)

`QEventSequence.__getitem__` (*expr*)

`QEventSequence.__iter__` ()

`QEventSequence.__len__` ()

(`AbjadObject`) `.__ne__` (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

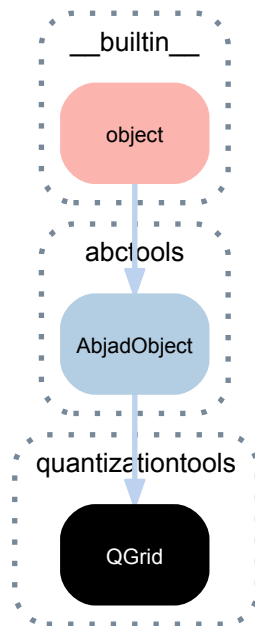
Returns boolean.

(`AbjadObject`) `.__repr__` ()

Interpreter representation of Abjad object.

Returns string.

### 21.2.19 quantizationtools.QGrid



**class** `quantizationtools.QGrid`(*root\_node=None*, *next\_downbeat=None*)

A rhythm-tree-based model for how millisecond attack points collapse onto the offsets generated by a nested rhythmic structure:

```
>>> q_grid = quantizationtools.QGrid()
```

```
>>> q_grid
quantizationtools.QGrid(
  root_node=quantizationtools.QGridLeaf(
    preprolated_duration=durationtools.Duration(1, 1),
    q_event_proxies=[],
    is_divisible=True
  ),
  next_downbeat=quantizationtools.QGridLeaf(
    preprolated_duration=durationtools.Duration(1, 1),
    q_event_proxies=[],
    is_divisible=True
  )
)
```

QGrids model not only the internal nodes of the nesting structure, but also the downbeat to the “next” QGrid, allowing events which occur very late within one structure to collapse virtually onto the beginning of the next structure.

QEventProxies can be “loaded in” to the node contained by the QGrid closest to their virtual offset:

```
>>> q_event_a = quantizationtools.PitchedQEvent(250, [0])
>>> q_event_b = quantizationtools.PitchedQEvent(750, [1])
>>> proxy_a = quantizationtools.QEventProxy(q_event_a, 0.25)
>>> proxy_b = quantizationtools.QEventProxy(q_event_b, 0.75)
```

```
>>> q_grid.fit_q_events([proxy_a, proxy_b])
```

```
>>> q_grid.root_node.q_event_proxies
[quantizationtools.QEventProxy(
  quantizationtools.PitchedQEvent(
    durationtools.Offset(250, 1),
    (NamedPitch("c'"),),
    attachments=()
  ),
  durationtools.Offset(1, 4)
)]
```

```
>>> q_grid.next_downbeat.q_event_proxies
[quantizationtools.QEventProxy(
  quantizationtools.PitchedQEvent(
    durationtools.Offset(750, 1),
    (NamedPitch("cs'"),),
    attachments=()
  ),
  durationtools.Offset(3, 4)
)]
```

Used internally by the Quantizer.

Return QGrid instance.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### **QGrid.distance**

The computed total distance of the offset of each `QEventProxy` contained by the `QGrid` to the offset of the `QGridLeaf` to which the `QEventProxy` is attached.

Return `Duration` instance.

### **QGrid.leaves**

All of the leaf nodes in the `QGrid`, including the next downbeat's node.

Returns tuple of `QGridLeaf` instances.

### **QGrid.next\_downbeat**

The node representing the “next” downbeat after the contents of the `QGrid`'s tree.

Return `QGridLeaf` instance.

### **QGrid.offsets**

The offsets between 0 and 1 of all of the leaf nodes in the `QGrid`.

Returns tuple of `Offset` instances.

### **QGrid.pretty\_rtm\_format**

The pretty RTM-format of the root node of the `QGrid`.

Returns string.

### **QGrid.root\_node**

The root node of the `QGrid`.

Return `QGridLeaf` or `QGridContainer`.

### **QGrid.rtm\_format**

The RTM format of the root node of the `QGrid`.

Returns string.

## Methods

### **QGrid.fit\_q\_events** (*q\_event\_proxies*)

Fit each `QEventProxy` in `q_event_proxies` onto the contained `QGridLeaf` whose offset is nearest.

Returns `None`

`QGrid.sort_q_events_by_index()`

Sort `QEventProxies` attached to each `QGridLeaf` in a `QGrid` by their index.

Returns `None`.

`QGrid.subdivide_leaf(leaf, subdivisions)`

Replace the `QGridLeaf` `leaf` contained in a `QGrid` by a `QGridContainer` containing `QGridLeaves` with durations equal to the ratio described in `subdivisions`

Returns the `QEventProxies` attached to `leaf`.

`QGrid.subdivide_leaves(pairs)`

Given a sequence of leaf-index:subdivision-ratio pairs `pairs`, subdivide the `QGridLeaves` described by the indices into `QGridContainers` containing `QGridLeaves` with durations equal to their respective subdivision-ratios.

Returns the `QEventProxies` attached to thus subdivided `QGridLeaf`.

## Special methods

`QGrid.__call__(beatspan)`

`QGrid.__copy__(*args)`

`QGrid.__deepcopy__(memo)`

`QGrid.__eq__(expr)`

`QGrid.__getstate__()`

`(AbjadObject).__ne__(expr)`

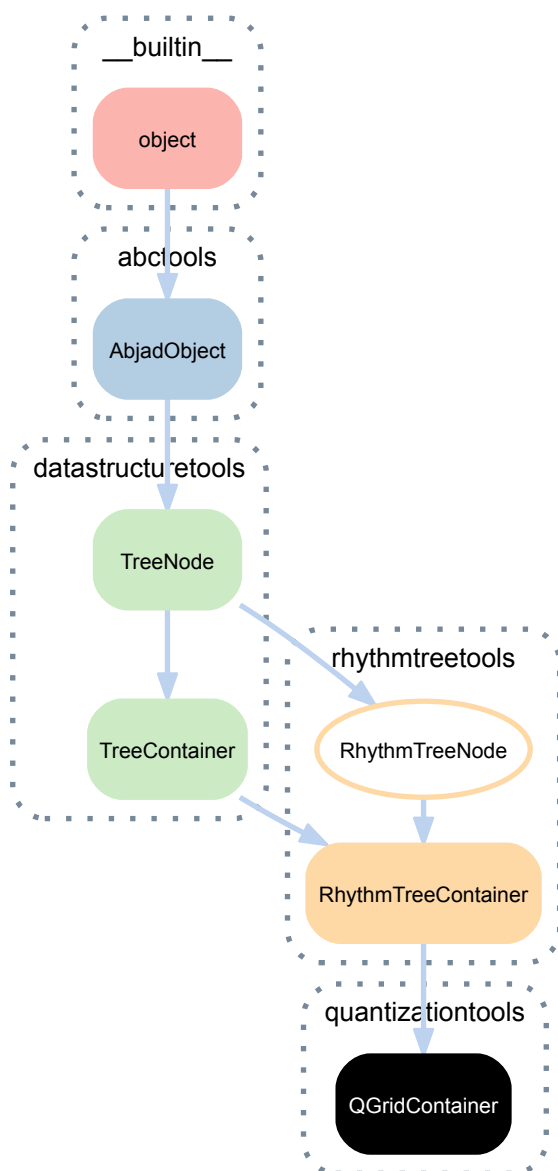
True when ID of `expr` does not equal ID of Abjad object.

Returns boolean.

`QGrid.__repr__()`

`QGrid.__setstate__(state)`

## 21.2.20 quantizationtools.QGridContainer



**class** `quantizationtools.QGridContainer` (*children=None*, *name=None*, *preprolated\_duration=1*,

A container in a QGrid structure:

```
>>> container = quantizationtools.QGridContainer()
```

```
>>> container
QGridContainer(
  preprolated_duration=Duration(1, 1)
)
```

Used internally by QGrid.

Return QGridContainer instance.

### Bases

- `rhythmtreetools.RhythmTreeContainer`
- `rhythmtreetools.RhythmTreeNode`



- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `(TreeContainer).children`

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

### `(TreeNode).depth`

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

### `(TreeNode).depthwise_inventory`

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

(RhythmTreeNode).**duration**

The prolated preprolated\_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

(TreeNode).**graph\_order**

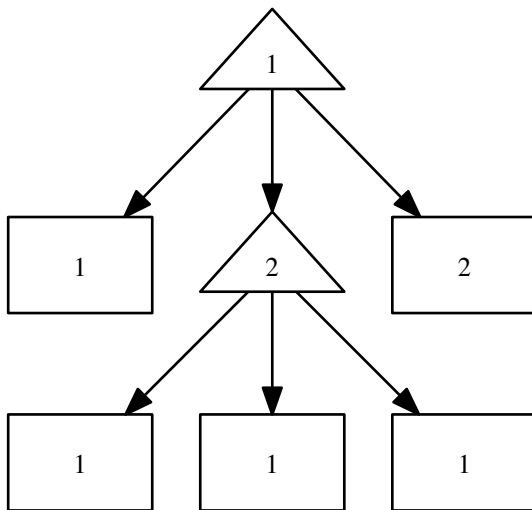
(RhythmTreeNode).**graphviz\_format**

(RhythmTreeContainer).**graphviz\_graph**

The GraphvizGraph representation of the RhythmTreeContainer:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> graph = tree.graphviz_graph
>>> print graph.graphviz_format
digraph G {
    node_0 [label=1,
            shape=triangle];
    node_1 [label=1,
            shape=box];
    node_2 [label=2,
            shape=triangle];
    node_3 [label=1,
            shape=box];
    node_4 [label=1,
            shape=box];
    node_5 [label=1,
            shape=box];
    node_6 [label=2,
            shape=box];
    node_0 -> node_1;
    node_0 -> node_2;
    node_0 -> node_6;
    node_2 -> node_3;
    node_2 -> node_4;
    node_2 -> node_5;
}
```

```
>>> iotools.graph(graph)
```



Return *GraphvizGraph* instance.

(*TreeNode*) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeContainer*) **.leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

(*TreeContainer*) **.nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(TreeNode) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(RhythmTreeNode) **.parentage\_ratios**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated\_duration of the root node, and subsequent items are pairs of the preprolated duration of the next node in the parentage chain and the total preprolated\_duration of that node and its siblings:

```
>>> a = rhythmtreertools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreertools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreertools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreertools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreertools.RhythmTreeLeaf(preprolated_duration=5)
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.parentage_ratios
(Duration(1, 1),)
```

```
>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))
```

```
>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))
```

```
>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))
```

```
>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Returns tuple.

(RhythmTreeNode).**.pretty\_rtm\_format**

The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
  (1 (
    1
    1))
  (1 (
    1
    1))))
```

Returns string.

(RhythmTreeNode).**.prolation**

(RhythmTreeNode).**.prolations**

(TreeNode).**.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeNode).**.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

(RhythmTreeContainer) **.rtm\_format**

The node's RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> tree.rtm_format
'(1 ((1 (1 1)) (1 (1 1))))'
```

Returns string.

(RhythmTreeNode) **.start\_offset**

The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Returns Offset instance.

(RhythmTreeNode) **.stop\_offset**

The stopping offset of a node in a rhythm-tree relative the root.

## Read/write properties

(TreeNode) **.name**

(RhythmTreeNode) **.preprolated\_duration**

The node's preprolated\_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(
...     preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Returns int.

## Methods

(TreeContainer) **.append(node)**

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.extend** (*expr*)  
Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.index** (*node*)  
Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) **.insert** (*i*, *node*)  
Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.pop** (*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) **.remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```



```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

(RhythmTreeContainer).**\_\_add\_\_**(*expr*)

Concatenate containers self and *expr*. The operation  $c = a + b$  returns a new RhythmTreeContainer *c* with the content of both *a* and *b*, and a *preprolated\_duration* equal to the sum of the durations of *a* and *b*. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand:

```
>>> a = rhythmtreetools.RhythmTreeParser() ('(1 (1 1 1))')[0]
>>> b = rhythmtreetools.RhythmTreeParser() ('(2 (3 4))')[0]
```

```
>>> c = a + b
```

```
>>> c.preprolated_duration
Duration(3, 1)
```

```
>>> c
RhythmTreeContainer(
  children=(
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(3, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(4, 1),
      is_pitched=True
    )
  ),
  preprolated_duration=Duration(3, 1)
)
```

Returns new RhythmTreeContainer.

(RhythmTreeContainer).**\_\_call\_\_**(*pulse\_duration*)

Generate Abjad score components:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree((1, 4))
[FixedDurationTuplet(1/4, [c'16, {@ 3:2 c'16, c'16, c'16 @}, c'8])]
```

Returns sequence of components.

(TreeContainer).**\_\_contains\_\_**(*expr*)  
 True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode).**\_\_copy\_\_**(\*args)

(TreeNode).**\_\_deepcopy\_\_**(\*args)

(TreeContainer).**\_\_delitem\_\_**(*i*)  
 Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(RhythmTreeContainer).**\_\_eq\_\_**(*expr*)  
 True if type, preprolated\_duration and children are equivalent. Otherwise False.

Returns boolean.

(TreeContainer).**\_\_getitem\_\_**(*i*)  
 Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

```
(TreeNode).__getstate__()
```

```
(TreeContainer).__iter__()
```

```
(TreeContainer).__len__()
```

Returns nonnegative integer number of nodes in container.

```
(TreeNode).__ne__(expr)
```

```
(TreeNode).__repr__()
```

```
(RhythmTreeContainer).__setitem__(i, expr)
```

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = rhythmtreetools.RhythmTreeContainer()
>>> b = rhythmtreetools.RhythmTreeLeaf()
>>> c = rhythmtreetools.RhythmTreeLeaf()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

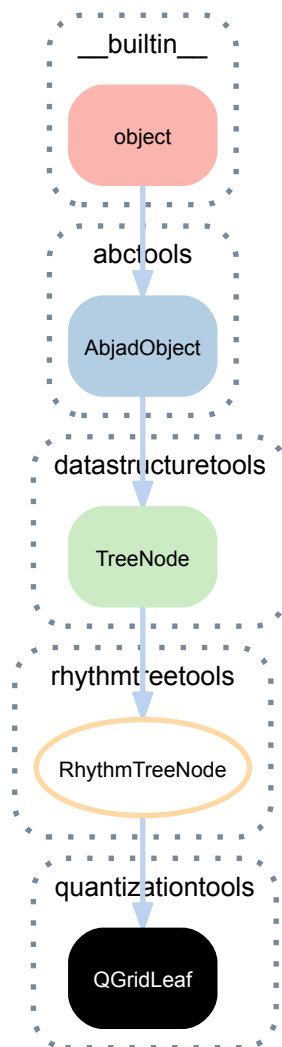
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

```
(TreeNode).__setstate__(state)
```

### 21.2.21 quantizationtools.QGridLeaf



**class** `quantizationtools.QGridLeaf` (*preprolated\_duration=1*, *q\_event\_proxies=None*, *is\_divisible=True*)

A leaf in a QGrid structure:

```
>>> leaf = quantizationtools.QGridLeaf()
```

```
>>> leaf
QGridLeaf(
  preprolated_duration=Duration(1, 1),
  is_divisible=True
)
```

Used internally by QGrid.

Return QGridLeaf instance.

#### Bases

- `rhythmtreetools.RhythmTreeNode`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### (TreeNode) .depth

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

### (TreeNode) .depthwise\_inventory

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

### (RhythmTreeNode) .duration

The prolated preprolated\_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreertools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

(TreeNode) **.graph\_order**

(RhythmTreeNode) **.graphviz\_format**

QGridLeaf **.graphviz\_graph**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeNode) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(RhythmTreeNode) **.parentage\_ratios**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated\_duration of the root node, and subsequent items are pairs of the preprolated duration of the next node in the parentage chain and the total preprolated\_duration of that node and its siblings:

```
>>> a = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.parentage_ratios
(Duration(1, 1),)
```

```
>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))
```

```
>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))
```

```
>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))
```

```
>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Returns tuple.

`QGridLeaf.preceding_q_event_proxies`

`(RhythmTreeNode).pretty_rtm_format`

The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
  (1 (
    1
  ))
  (1 (
    1
  )))
)
```

Returns string.

`(RhythmTreeNode).prolation`

`(RhythmTreeNode).prolations`

`(TreeNode).proper_parentage`

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

`QGridLeaf.q_event_proxies`

`(TreeNode).root`

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

**QGridLeaf.rtm\_format**

(RhythmTreeNode).**start\_offset**

The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Returns Offset instance.

(RhythmTreeNode).**stop\_offset**

The stopping offset of a node in a rhythm-tree relative the root.

**QGridLeaf.succeeding\_q\_event\_proxies**

## Read/write properties

**QGridLeaf.is\_divisible**

Flag for whether the node may be further divided under some search tree.

(TreeNode).**name**

(RhythmTreeNode).**preprolated\_duration**

The node's preprolated\_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(
...     preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Returns int.

## Special methods

**QGridLeaf.\_\_call\_\_**(pulse\_duration)

(TreeNode).**\_\_copy\_\_**(\*args)

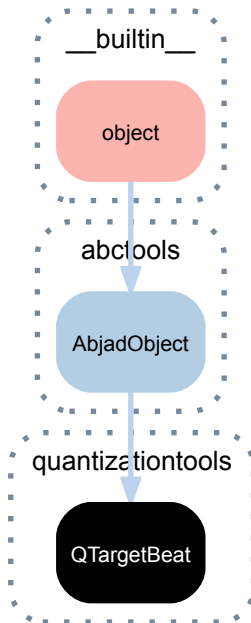
**QGridLeaf.\_\_deepcopy\_\_**(memo)

**QGridLeaf.\_\_eq\_\_**(expr)



```
(TreeNode) .__getstate__()
(TreeNode) .__ne__(expr)
(TreeNode) .__repr__()
(TreeNode) .__setstate__(state)
```

### 21.2.22 quantizationtools.QTargetBeat



```
class quantizationtools.QTargetBeat (beatspan=None, offset_in_ms=None,
                                     search_tree=None, tempo=None)
```

Representation of a single “beat” in a quantization target:

```
>>> beatspan = (1, 8)
>>> offset_in_ms = 1500
>>> search_tree = quantizationtools.UnweightedSearchTree({3: None})
>>> tempo = contexttools.TempoMark((1, 4), 56)
```

```
>>> q_target_beat = quantizationtools.QTargetBeat (
...     beatspan=beatspan,
...     offset_in_ms=offset_in_ms,
...     search_tree=search_tree,
...     tempo=tempo,
... )
```

```
>>> q_target_beat
quantizationtools.QTargetBeat (
  beatspan=durationtools.Duration(1, 8),
  offset_in_ms=durationtools.Offset(1500, 1),
  search_tree=quantizationtools.UnweightedSearchTree (
    definition={ 3: None}
  ),
  tempo=contexttools.TempoMark (
    durationtools.Duration(1, 4),
    56
  )
)
```

Not composer-safe.

Used internally by quantizationtools.Quantizer.

Return QTargetBeat instance.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `QTargetBeat.beatspan`

The beatspan of the `QTargetBeat`:

```
>>> q_target_beat.beatspan
Duration(1, 8)
```

Returns `Duration`.

### `QTargetBeat.distances`

A list of computed distances between the `QEventProxies` associated with a `QTargetBeat` instance, and each `QGrid` generated for that beat.

Used internally by the `Quantizer`.

Returns tuple.

### `QTargetBeat.duration_in_ms`

The duration in milliseconds of the `QTargetBeat`:

```
>>> q_target_beat.duration_in_ms
Duration(3750, 7)
```

Returns `Duration` instance.

### `QTargetBeat.offset_in_ms`

The offset in milliseconds of the `QTargetBeat`:

```
>>> q_target_beat.offset_in_ms
Offset(1500, 1)
```

Returns `Offset` instance.

### `QTargetBeat.q_events`

A list for storing `QEventProxy` instances.

Used internally by the `Quantizer`.

Returns list.

### `QTargetBeat.q_grid`

The `QGrid` instance selected by a `Heuristic`.

Used internally by the `Quantizer`.

Return `QGrid` instance.

### `QTargetBeat.q_grids`

A tuple of `QGrids` generated by a `QuantizationJob`.

Used internally by the `Quantizer`.

Returns tuple.

### `QTargetBeat.search_tree`

The search tree of the `QTargetBeat`:

```
>>> q_target_beat.search_tree
UnweightedSearchTree(
  definition={ 3: None}
)
```

Return `SearchTree` instance.

`QTargetBeat`.**tempo**

The tempo of the `QTargetBeat`:

```
>>> q_target_beat.tempo
TempoMark(Duration(1, 4), 56)
```

Return `TempoMark` instance.

## Special methods

`QTargetBeat`.**\_\_call\_\_**(*job\_id*)

(`AbjadObject`).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

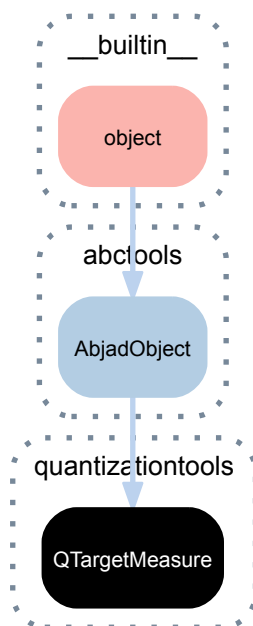
(`AbjadObject`).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`QTargetBeat`.**\_\_repr\_\_**()

### 21.2.23 quantizationtools.QTargetMeasure



```
class quantizationtools.QTargetMeasure (offset_in_ms=None,           search_tree=None,
                                         time_signature=None,         tempo=None,
                                         use_full_measure=False)
```

Representation of a single “measure” in a measure-wise quantization target:

```
>>> search_tree = quantizationtools.UnweightedSearchTree({2: None})
>>> tempo = contexttools.TempoMark((1, 4), 60)
>>> time_signature = contexttools.TimeSignatureMark((4, 4))
```

```
>>> q_target_measure = quantizationtools.QTargetMeasure(
...     offset_in_ms=1000,
...     search_tree=search_tree,
...     tempo=tempo,
```

```
...     time_signature=time_signature,
...     )
```

```
>>> q_target_measure
quantizationtools.QTargetMeasure(
  offset_in_ms=durationtools.Offset(1000, 1),
  search_tree=quantizationtools.UnweightedSearchTree(
    definition={ 2: None}
  ),
  time_signature=contexttools.TimeSignatureMark(
    (4, 4)
  ),
  tempo=contexttools.TempoMark(
    durationtools.Duration(1, 4),
    60
  ),
  use_full_measure=False
)
```

QTargetMeasures group QTargetBeats:

```
>>> for q_target_beat in q_target_measure.beats:
...     print q_target_beat.offset_in_ms, q_target_beat.duration_in_ms
1000 1000
2000 1000
3000 1000
4000 1000
```

If `use_full_measure` is set, the `QTargetMeasure` will only ever contain a single `QTargetBeat` instance:

```
>>> another_q_target_measure = quantizationtools.QTargetMeasure(
...     offset_in_ms=1000,
...     search_tree=search_tree,
...     tempo=tempo,
...     time_signature=time_signature,
...     use_full_measure=True,
...     )
```

```
>>> for q_target_beat in another_q_target_measure.beats:
...     print q_target_beat.offset_in_ms, q_target_beat.duration_in_ms
1000 4000
```

Not composer-safe.

Used internally by `Quantizer`.

Return `QTargetMeasure` instance.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`QTargetMeasure.beats`

The tuple of `QTargetBeats` contained by the `QTargetMeasure`:

```
>>> for q_target_beat in q_target_measure.beats:
...     q_target_beat
quantizationtools.QTargetBeat(
  beatspan=durationtools.Duration(1, 4),
  offset_in_ms=durationtools.Offset(1000, 1),
  search_tree=quantizationtools.UnweightedSearchTree(
    definition={ 2: None}
  ),
)
```

```

        tempo=contextttools.TempoMark(
            durationtools.Duration(1, 4),
            60
        )
    )
    quantizationtools.QTargetBeat(
        beatspan=durationtools.Duration(1, 4),
        offset_in_ms=durationtools.Offset(2000, 1),
        search_tree=quantizationtools.UnweightedSearchTree(
            definition={ 2: None}
        ),
        tempo=contextttools.TempoMark(
            durationtools.Duration(1, 4),
            60
        )
    )
    quantizationtools.QTargetBeat(
        beatspan=durationtools.Duration(1, 4),
        offset_in_ms=durationtools.Offset(3000, 1),
        search_tree=quantizationtools.UnweightedSearchTree(
            definition={ 2: None}
        ),
        tempo=contextttools.TempoMark(
            durationtools.Duration(1, 4),
            60
        )
    )
    quantizationtools.QTargetBeat(
        beatspan=durationtools.Duration(1, 4),
        offset_in_ms=durationtools.Offset(4000, 1),
        search_tree=quantizationtools.UnweightedSearchTree(
            definition={ 2: None}
        ),
        tempo=contextttools.TempoMark(
            durationtools.Duration(1, 4),
            60
        )
    )
)

```

Returns tuple.

#### **QTargetMeasure.duration\_in\_ms**

The duration in milliseconds of the QTargetMeasure:

```

>>> q_target_measure.duration_in_ms
Duration(4000, 1)

```

Returns Duration.

#### **QTargetMeasure.offset\_in\_ms**

The offset in milliseconds of the QTargetMeasure:

```

>>> q_target_measure.offset_in_ms
Offset(1000, 1)

```

Returns Offset.

#### **QTargetMeasure.search\_tree**

The search tree of the QTargetMeasure:

```

>>> q_target_measure.search_tree
UnweightedSearchTree(
    definition={ 2: None}
)

```

Return SearchTree instance.

#### **QTargetMeasure.tempo**

The tempo of the QTargetMeasure:

```
>>> q_target_measure.tempo
TempoMark(Duration(1, 4), 60)
```

Return TempoMark instance.

**QTargetMeasure.time\_signature**

The time signature of the QTargetMeasure:

```
>>> q_target_measure.time_signature
TimeSignatureMark((4, 4))
```

Return TimeSignatureMark instance.

**QTargetMeasure.use\_full\_measure**

The use\_full\_measure flag of the QTargetMeasure:

```
>>> q_target_measure.use_full_measure
False
```

Returns boolean.

## Special methods

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

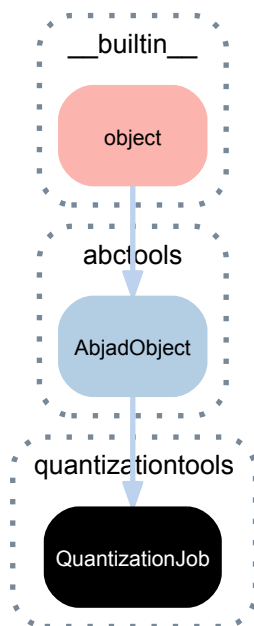
(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

QTargetMeasure.**\_\_repr\_\_**()

## 21.2.24 quantizationtools.QuantizationJob



**class** quantizationtools.**QuantizationJob**(*job\_id*, *search\_tree*, *q\_event\_proxies*,  
*q\_grids=None*)

A copiable, picklable class for generating all QGrids which are valid under a given SearchTree for a sequence of QEventProxies:

```

>>> q_event_a = quantizationtools.PitchedQEvent(250, [0, 1])
>>> q_event_b = quantizationtools.SilentQEvent(500)
>>> q_event_c = quantizationtools.PitchedQEvent(750, [3, 7])
>>> proxy_a = quantizationtools.QEventProxy(q_event_a, 0.25)
>>> proxy_b = quantizationtools.QEventProxy(q_event_b, 0.5)
>>> proxy_c = quantizationtools.QEventProxy(q_event_c, 0.75)

>>> definition = {2: {2: None}, 3: None, 5: None}
>>> search_tree = quantizationtools.UnweightedSearchTree(definition)

>>> job = quantizationtools.QuantizationJob(
...     1, search_tree, [proxy_a, proxy_b, proxy_c])

```

QuantizationJob generates QGrids when called, and stores those QGrids on its `q_grids` attribute, allowing them to be recalled later, even if pickled:

```

>>> job()
>>> for q_grid in job.q_grids:
...     print q_grid.rtm_format
1
(1 (1 1 1 1 1))
(1 (1 1 1))
(1 (1 1))
(1 ((1 (1 1)) (1 (1 1))))

```

QuantizationJob is intended to be useful in multiprocessing-enabled environments.

Return QuantizationJob instance.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### QuantizationJob.job\_id

The job id of the QuantizationJob:

```

>>> job.job_id
1

```

Only meaningful when the job is processed via multiprocessing, as the job id is necessary to reconstruct the order of jobs.

Returns int.

### QuantizationJob.q\_event\_proxies

The QEventProxies the QuantizationJob was instantiated with:

```

>>> for q_event_proxy in job.q_event_proxies:
...     q_event_proxy
quantizationtools.QEventProxy(
    quantizationtools.PitchedQEvent(
        durationtools.Offset(250, 1),
        (NamedPitch("c'"), NamedPitch("cs'")),
        attachments=()
    ),
    durationtools.Offset(1, 4)
)
quantizationtools.QEventProxy(
    quantizationtools.SilentQEvent(
        durationtools.Offset(500, 1),
        attachments=()
    ),
    durationtools.Offset(1, 2)
)

```

```

    )
    quantizationtools.QEventProxy(
        quantizationtools.PitchedQEvent(
            durationtools.Offset(750, 1),
            (NamedPitch("ef"), NamedPitch("g")),
            attachments=()
        ),
        durationtools.Offset(3, 4)
    )

```

Returns tuple.

`QuantizationJob.q_grids`

The generated QGrids:

```

>>> for q_grid in job.q_grids:
...     print q_grid.rtm_format
1
(1 (1 1 1 1 1))
(1 (1 1 1))
(1 (1 1))
(1 ((1 (1 1)) (1 (1 1))))

```

Returns tuple.

`QuantizationJob.search_tree`

The search tree the `QuantizationJob` was instantiated with:

```

>>> job.search_tree
UnweightedSearchTree(
  definition={ 2: { 2: None}, 3: None, 5: None}
)

```

Return `SearchTree` instance.

## Special methods

`QuantizationJob.__call__()`

`QuantizationJob.__eq__(expr)`

`QuantizationJob.__getstate__()`

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(AbjadObject).__repr__()`

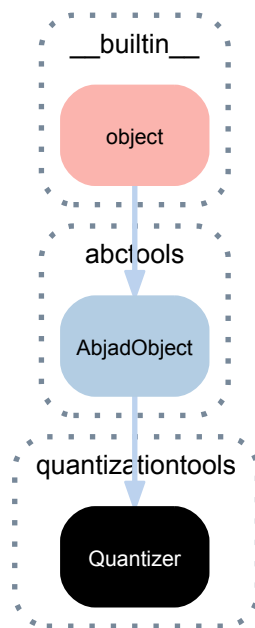
Interpreter representation of Abjad object.

Returns string.

`QuantizationJob.__setstate__(state)`



### 21.2.25 quantizationtools.Quantizer



**class** `quantizationtools.Quantizer`

`Quantizer` quantizes sequences of attack-points, encapsulated by `QEventSequences`, into score trees:

```
>>> quantizer = quantizationtools.Quantizer()
```

```
>>> durations = [1000] * 8
>>> pitches = range(8)
>>> q_event_sequence = \
...     quantizationtools.QEventSequence.from_millisecond_pitch_pairs(
...     zip(durations, pitches))
```

Quantization defaults to outputting into a 4/4, quarter=60 musical structure:

```
>>> result = quantizer(q_event_sequence)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
  \new Staff {
    \new Voice {
      {
        \time 4/4
        \tempo 4=60
        c'4
        cs'4
        d'4
        ef'4
      }
      {
        e'4
        f'4
        fs'4
        g'4
      }
    }
  }
>>
```

```
>>> show(score)
```



However, the behavior of the `Quantizer` can be modified at call-time. Passing a `QSchema` instance will alter the macro-structure of the output.

Here, we quantize using settings specified by a `MeasurewiseQSchema`, which will cause the `Quantizer` to group the output into measures with different tempi and time signatures:

```
>>> measurewise_q_schema = quantizationtools.MeasurewiseQSchema(
...     {'tempo': ((1, 4), 78), 'time_signature': (2, 4)},
...     {'tempo': ((1, 8), 57), 'time_signature': (5, 4)},
... )

>>> result = quantizer(q_event_sequence, q_schema=measurewise_q_schema)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
  \new Staff {
    \new Voice {
      {
        \time 2/4
        \tempo 4=78
        c'4 ~
        \times 4/5 {
          c'16.
          cs'8.. ~
        }
      }
      {
        \time 5/4
        \tempo 8=57
        \times 4/7 {
          cs'16.
          d'8 ~
        }
        \times 4/5 {
          d'16
          ef'16. ~
        }
        \times 2/3 {
          ef'16
          e'8 ~
        }
        \times 4/7 {
          e'16
          f'8 ~
          f'32 ~
        }
        f'32
        fs'16. ~
        \times 4/5 {
          fs'32
          g'8 ~
        }
        \times 4/7 {
          g'32
          r4. ~
          r32 ~
        }
      }
    }
  }
>>
```

```
>>> show(score)
```



Here we quantize using settings specified by a `BeatwiseQSchema`, which keeps the output of the quantizer “flattened”, without measures or explicit time signatures. The default beat-wise settings of `quarter=60` persists until the third “beatspan”:

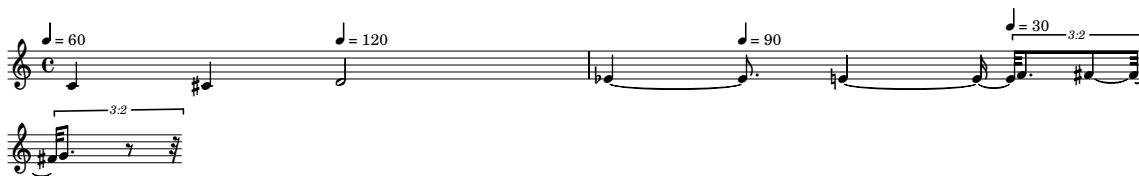
```

>>> beatwise_q_schema = quantizationtools.BeatwiseQSchema(
... {
...     2: {'tempo': ((1, 4), 120)},
...     5: {'tempo': ((1, 4), 90)},
...     7: {'tempo': ((1, 4), 30)},
... })

>>> result = quantizer(q_event_sequence, q_schema=beatwise_q_schema)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
  \new Staff {
    \new Voice {
      \tempo 4=60
      c'4
      cs'4
      \tempo 4=120
      d'2
      ef'4 ~
      \tempo 4=90
      ef'8.
      e'4 ~
      e'16 ~
      \times 2/3 {
        \tempo 4=30
        e'32
        f'8.
        fs'8 ~
        fs'32 ~
      }
      \times 2/3 {
        fs'32
        g'8.
        r8 ~
        r32
      }
    }
  }
>>

```

```
>>> show(score)
```



Note that TieChains are generally fused together in the above example, but break at tempo changes.

Other keyword arguments are:

- `grace_handler`: a `GraceHandler` instance controls whether and how grace notes are used in the output. Options currently include `CollapsingGraceHandler`, `ConcatenatingGraceHandler` and `DiscardingGraceHandler`.
- `heuristic`: a `Heuristic` instance controls how output rhythms are selected from a pool of candidates. Options currently include the `DistanceHeuristic` class.
- `job_handler`: a `JobHandler` instance controls whether or not parallel processing is used during the quantization process. Options include the `SerialJobHandler` and `ParallelJobHandler` classes.
- `attack_point_optimizer`: an `AttackPointOptimizer` instance controls whether and how tie chains are re-notated. Options currently include `MeasurewiseAttackPointOptimizer`, `NaiveAttackPointOptimizer` and `NullAttackPointOptimizer`.

Refer to the reference pages for `BeatwiseQSchema` and `MeasurewiseQSchema` for more informa-

tion on controlling the `Quantizer`'s output, and to the reference on `SearchTree` for information on controlling the rhythmic complexity of that same output.

Return `Quantizer` instance.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Special methods

`Quantizer.__call__(q_event_sequence, q_schema=None, grace_handler=None, heuristic=None, job_handler=None, attack_point_optimizer=None, attach_tempo_marks=True)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

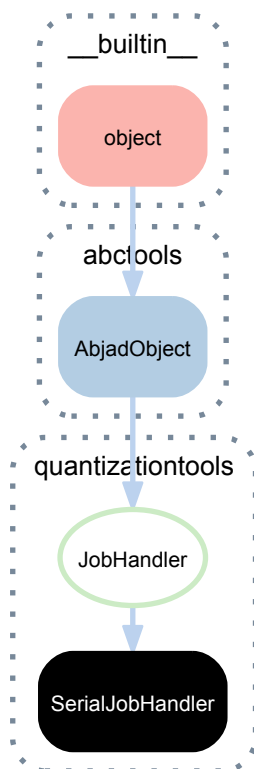
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 21.2.26 quantizationtools.SerialJobHandler



**class** `quantizationtools.SerialJobHandler`  
Processes `QuantizationJob` instances sequentially.

### Bases

- `quantizationtools.JobHandler`
- `abctools.AbjadObject`
- `__builtin__.object`

### Special methods

`SerialJobHandler.__call__(jobs)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

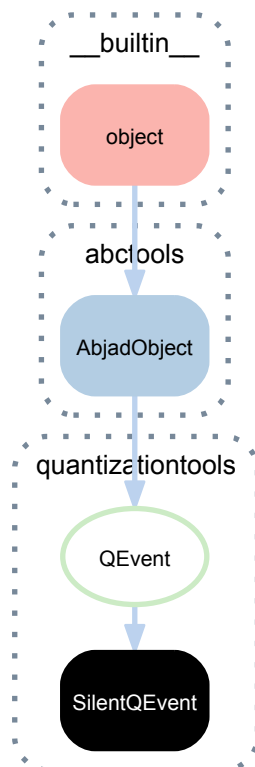
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 21.2.27 quantizationtools.SilentQEvent



**class** `quantizationtools.SilentQEvent` (*offset*, *attachments=None*, *index=None*)  
A `QEvent` which indicates the onset of a period of silence in a `QEventSequence`:

```
>>> q_event = quantizationtools.SilentQEvent(1000)
>>> q_event
quantizationtools.SilentQEvent(
  durationtools.Offset(1000, 1),
  attachments=()
)
```

Return `SilentQEvent` instance.

## Bases

- `quantizationtools.QEvent`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`SilentQEvent.attachments`

`(QEvent).index`

The optional index, for sorting QEvents with identical offsets.

`(QEvent).offset`

The offset in milliseconds of the event.

`(QEvent).storage_format`

Storage format of q-event.

Returns string.

## Special methods

`SilentQEvent.__eq__(expr)`

`(QEvent).__getstate__()`

`(QEvent).__lt__(expr)`

`(AbjadObject).__ne__(expr)`

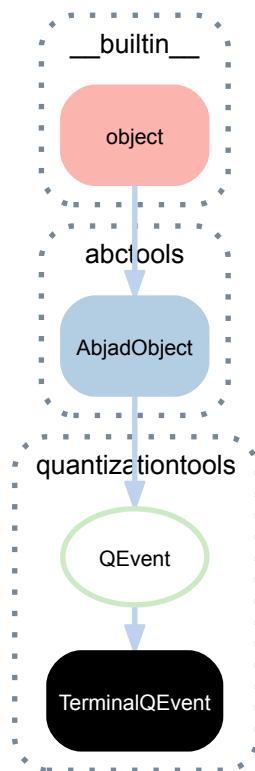
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(QEvent).__repr__()`

`(QEvent).__setstate__(state)`

### 21.2.28 quantizationtools.TerminalQEvent



**class** `quantizationtools.TerminalQEvent` (*offset*)

The terminal event in a series of QEvents:

```

>>> q_event = quantizationtools.TerminalQEvent(1000)
>>> q_event
quantizationtools.TerminalQEvent(
    durationtools.Offset(1000, 1)
)
  
```

Carries no significance outside the context of a QEventSequence.

Return `TerminalQEvent` instance.

#### Bases

- `quantizationtools.QEvent`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(QEvent).index`

The optional index, for sorting QEvents with identical offsets.

`(QEvent).offset`

The offset in milliseconds of the event.

`(QEvent).storage_format`

Storage format of q-event.

Returns string.

## Special methods

`TerminalQEvent.__eq__(expr)`

`(QEvent).__getstate__()`

`(QEvent).__lt__(expr)`

`(AbjadObject).__ne__(expr)`

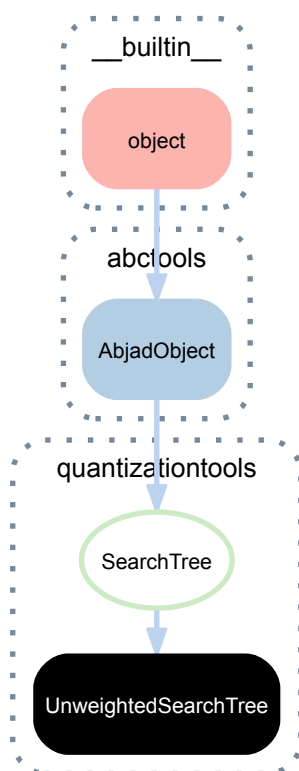
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(QEvent).__repr__()`

`(QEvent).__setstate__(state)`

### 21.2.29 quantizationtools.UnweightedSearchTree



**class** `quantizationtools.UnweightedSearchTree` (*definition=None*)

Concrete `SearchTree` subclass, based on Paul Nauert's search tree model:

```

>>> search_tree = quantizationtools.UnweightedSearchTree()
>>> search_tree
UnweightedSearchTree(
  definition={ 2: { 2: { 2: { 2: None}, 3: None}, 3: None, 5: None, 7: None},
    3: { 2: { 2: None}, 3: None, 5: None},
    5: { 2: None, 3: None},
    7: { 2: None},
    11: None,
    13: None}
)
  
```

The search tree defines how nodes in a `QGrid` may be subdivided, if they happen to contain `QEvents` (or, in actuality, `QEventProxy` instances which reference `QEvents`, but rescale their offsets between 0 and 1).



In the default definition, the root node of the `QGrid` may be subdivided into 2, 3, 5, 7, 11 or 13 equal parts. If divided into 2 parts, the divisions of the root node may be divided again into 2, 3, 5 or 7, and so forth.

This definition is structured as a collection of nested dictionaries, whose keys are integers, and whose values are either the sentinel `None` indicating no further permissible divisions, or dictionaries obeying these same rules, which then indicate the possibilities for further division.

Calling a `UnweightedSearchTree` with a `QGrid` instance will generate all permissible subdivided `QGrids`, according to the definition of the called search tree:

```
>>> q_event_a = quantizationtools.PitchedQEvent(130, [0, 1, 4])
>>> q_event_b = quantizationtools.PitchedQEvent(150, [2, 3, 5])
>>> proxy_a = quantizationtools.QEventProxy(q_event_a, 0.5)
>>> proxy_b = quantizationtools.QEventProxy(q_event_b, 0.667)
>>> q_grid = quantizationtools.QGrid()
>>> q_grid.fit_q_events([proxy_a, proxy_b])
```

```
>>> q_grids = search_tree(q_grid)
>>> for grid in q_grids:
...     print grid.rtm_format
(1 (1 1))
(1 (1 1 1))
(1 (1 1 1 1 1))
(1 (1 1 1 1 1 1 1))
(1 (1 1 1 1 1 1 1 1 1 1))
(1 (1 1 1 1 1 1 1 1 1 1 1))
```

A custom `UnweightedSearchTree` may be defined by passing in a dictionary, as described above. The following search tree only permits divisions of the root node into 2s and 3s, and if divided into 2, a node may be divided once more into 2 parts:

```
>>> definition = {2: {2: None}, 3: None}
>>> search_tree = quantizationtools.UnweightedSearchTree(definition)
```

```
>>> q_grids = search_tree(q_grid)
>>> for grid in q_grids:
...     print grid.rtm_format
(1 (1 1))
(1 (1 1 1))
```

Return `UnweightedSearchTree` instance.

## Bases

- `quantizationtools.SearchTree`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `UnweightedSearchTree.default_definition`

The default search tree definition, based on the search tree given by Paul Nauert:

```
>>> import pprint
>>> search_tree = quantizationtools.UnweightedSearchTree()
>>> pprint.pprint(search_tree.default_definition)
{2: {2: {2: {2: None}, 3: None}, 3: None, 5: None, 7: None},
 3: {2: {2: None}, 3: None, 5: None},
 5: {2: None, 3: None},
 7: {2: None},
11: None,
13: None}
```

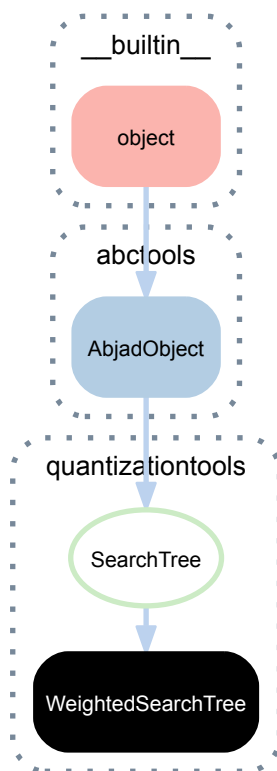
Returns dictionary.

(SearchTree).**definition**  
 The search tree definition.  
 Returns dictionary.

### Special methods

(SearchTree).**\_\_call\_\_**(*q\_grid*)  
 (SearchTree).**\_\_eq\_\_**(*expr*)  
 (SearchTree).**\_\_getstate\_\_**()  
 (AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.  
 (SearchTree).**\_\_repr\_\_**()  
 (SearchTree).**\_\_setstate\_\_**(*state*)

### 21.2.30 quantizationtools.WeightedSearchTree



**class** quantizationtools.**WeightedSearchTree** (*definition=None*)  
 A search tree that allows for dividing nodes in a QGrid into parts with unequal weights:

```

>>> search_tree = quantizationtools.WeightedSearchTree()

>>> search_tree
WeightedSearchTree(
  definition={
    'divisors': (2, 3, 5, 7),
    'max_depth': 3,
    'max_divisions': 2}
)
    
```

In `WeightedSearchTree`'s definition:

- `divisors` controls the sum of the parts of the ratio a node may be divided into,
- `max_depth` controls how many levels of tuple nesting are permitted, and
- `max_divisions` controls the maximum permitted length of the weights in the ratio.

Thus, the default `WeightedSearchTree` permits the following ratios:

```
>>> for x in search_tree.all_compositions:
...     x
...
(1, 1)
(2, 1)
(1, 2)
(4, 1)
(3, 2)
(2, 3)
(1, 4)
(6, 1)
(5, 2)
(4, 3)
(3, 4)
(2, 5)
(1, 6)
```

Return `WeightedSearchTree` instance.

## Bases

- `quantizationtools.SearchTree`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`WeightedSearchTree.all_compositions`

`WeightedSearchTree.default_definition`

`(SearchTree).definition`

The search tree definition.

Returns dictionary.

## Special methods

`(SearchTree).__call__(q_grid)`

`(SearchTree).__eq__(expr)`

`(SearchTree).__getstate__()`

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(SearchTree).__repr__()`

`(SearchTree).__setstate__(state)`

## 21.3 Functions

### 21.3.1 quantizationtools.make\_test\_time\_segments

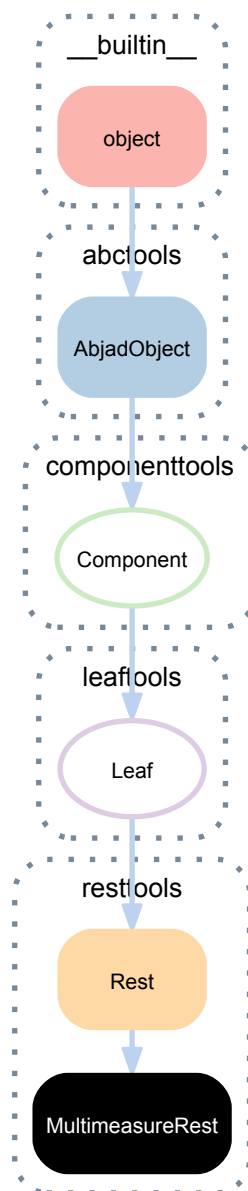
`quantizationtools.make_test_time_segments()`

Make test time segments.

# RESTTOOLS

## 22.1 Concrete classes

### 22.1.1 resttools.MultimeasureRest



**class** `resttools.MultimeasureRest` (*\*args, \*\*kwargs*)  
 Abjad model of a multi-measure rest:

```
>>> resttools.MultimeasureRest((1, 4))
MultimeasureRest('R4')
```

Multi-measure rests are immutable.

## Bases

- `resttools.Rest`
- `leaftools.Leaf`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Component).lilypond_format`  
 Lilypond format of component.  
 Returns string.

`(Component).override`  
 LilyPond grob override component plug-in.  
 Returns LilyPond grob override component plug-in.

`(Component).set`  
 LilyPond context setting component plug-in.  
 Returns LilyPond context setting component plug-in.

`(Component).storage_format`  
 Storage format of component.  
 Returns string.

## Read/write properties

`(Leaf).lilypond_duration_multiplier`  
 LilyPond duration multiplier.  
 Set to positive multiplier or none.  
 Returns positive multiplier or none.

`(Leaf).written_duration`  
 Written duration of leaf.  
 Set to duration.  
 Returns duration.

`(Leaf).written_pitch_indication_is_at_sounding_pitch`  
 Returns true when written pitch is at sounding pitch. Returns false when written pitch is transposed.

`(Leaf).written_pitch_indication_is_nonsemantic`  
 Returns true when pitch is nonsemantic. Returns false otherwise.  
 Set to true when using leaves only graphically.  
 Setting this value to true sets sounding pitch indicator to false.

## Methods

(Component) .**select** (*sequential=False*)  
Selects component.  
Returns component selection when *sequential* is false.  
Returns sequential selection when *sequential* is true.

## Special methods

(Component) .**\_\_copy\_\_** (\*args)  
Copies component with marks but without children of component or spanners attached to component.  
Returns new component.

(AbjadObject) .**\_\_eq\_\_** (expr)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(Component) .**\_\_mul\_\_** (n)  
Copies component *n* times and detaches spanners.  
Returns list of new components.

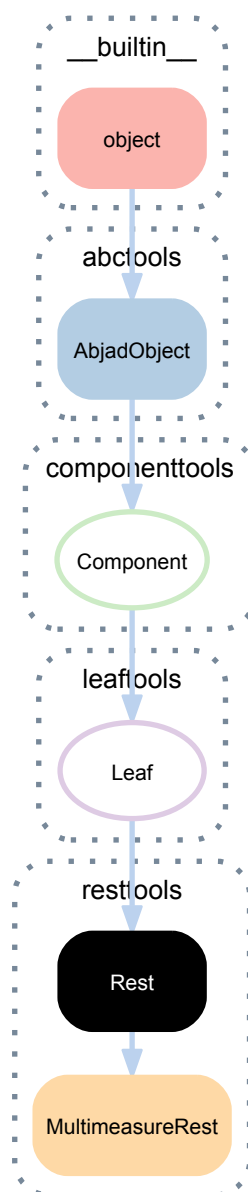
(AbjadObject) .**\_\_ne\_\_** (expr)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(Leaf) .**\_\_repr\_\_** ()  
Interpreter representation of leaf.  
Returns string.

(Component) .**\_\_rmul\_\_** (n)  
Copies component *n* times and detach spanners.  
Returns list of new components.

(Leaf) .**\_\_str\_\_** ()  
String representation of leaf.  
Returns string.

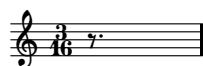
## 22.1.2 resttools.Rest



**class** `resttools.Rest` (\*args, \*\*kwargs)  
A rest.

### Example.

```
>>> rest = Rest('r8.')
>>> measure = Measure((3, 16), [rest])
>>> show(measure)
```



### Bases

- `leaftools.Leaf`
- `componenttools.Component`
- `abctools.AbjadObject`



- `__builtin__.object`

## Read-only properties

- (Component) **.lilypond\_format**  
Lilypond format of component.  
Returns string.
- (Component) **.override**  
LilyPond grob override component plug-in.  
Returns LilyPond grob override component plug-in.
- (Component) **.set**  
LilyPond context setting component plug-in.  
Returns LilyPond context setting component plug-in.
- (Component) **.storage\_format**  
Storage format of component.  
Returns string.

## Read/write properties

- (Leaf) **.lilypond\_duration\_multiplier**  
LilyPond duration multiplier.  
Set to positive multiplier or none.  
Returns positive multiplier or none.
- (Leaf) **.written\_duration**  
Written duration of leaf.  
Set to duration.  
Returns duration.
- (Leaf) **.written\_pitch\_indication\_is\_at\_sounding\_pitch**  
Returns true when written pitch is at sounding pitch. Returns false when written pitch is transposed.
- (Leaf) **.written\_pitch\_indication\_is\_nonsemantic**  
Returns true when pitch is nonsemantic. Returns false otherwise.  
Set to true when using leaves only graphically.  
Setting this value to true sets sounding pitch indicator to false.

## Methods

- (Component) **.select** (*sequential=False*)  
Selects component.  
Returns component selection when *sequential* is false.  
Returns sequential selection when *sequential* is true.

## Special methods

- (Component) **\_\_copy\_\_** (*\*args*)  
Copies component with marks but without children of component or spanners attached to component.  
Returns new component.

(AbjadObject) .**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(Component) .**\_\_mul\_\_**(*n*)  
 Copies component *n* times and detaches spanners.  
 Returns list of new components.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Leaf) .**\_\_repr\_\_**()  
 Interpreter representation of leaf.  
 Returns string.

(Component) .**\_\_rmul\_\_**(*n*)  
 Copies component *n* times and detach spanners.  
 Returns list of new components.

(Leaf) .**\_\_str\_\_**()  
 String representation of leaf.  
 Returns string.

## 22.2 Functions

### 22.2.1 resttools.make\_multimeasure\_rests

resttools.**make\_multimeasure\_rests**(*durations*)  
 Make multi-measure rests from *durations*:

```
>>> resttools.make_multimeasure_rests([(4, 4), (7, 4)])
Selection(MultimeasureRest('R1'), MultimeasureRest('R1..'))
```

Returns list.

### 22.2.2 resttools.make\_repeated\_rests\_from\_time\_signatures

resttools.**make\_repeated\_rests\_from\_time\_signatures**(*time\_signatures*)  
 Make repeated rests from *time\_signatures*:

```
resttools.make_repeated_rests_from_time_signatures([(2, 8), (3, 32)])
[[Rest('r8'), Rest('r8')], [Rest('r32'), Rest('r32'), Rest('r32')]]
```

Returns two-dimensional list of newly constructed rest lists.

Use `sequencetools.flatten_sequence()` to flatten output if required.

### 22.2.3 resttools.make\_rests

resttools.**make\_rests**(*durations*, *decrease\_durations\_monotonically*=True, *tie\_parts*=False)  
 Make rests.

Make rests and decrease durations monotonically:

```
>>> resttools.make_rests(
...     [(5, 16), (9, 16)],
...     decrease_durations_monotonically=True,
...     )
Selection(Rest('r4'), Rest('r16'), Rest('r2'), Rest('r16'))
```

Makes rests and increase durations monotonically:

```
>>> resttools.make_rests(
...     [(5, 16), (9, 16)],
...     decrease_durations_monotonically=False,
...     )
Selection(Rest('r16'), Rest('r4'), Rest('r16'), Rest('r2'))
```

Make tied rests:

```
>>> voice = Voice(resttools.make_rests(
...     [(5, 16), (9, 16)],
...     tie_parts=True,
...     ))
```

```
>>> show(voice)
```



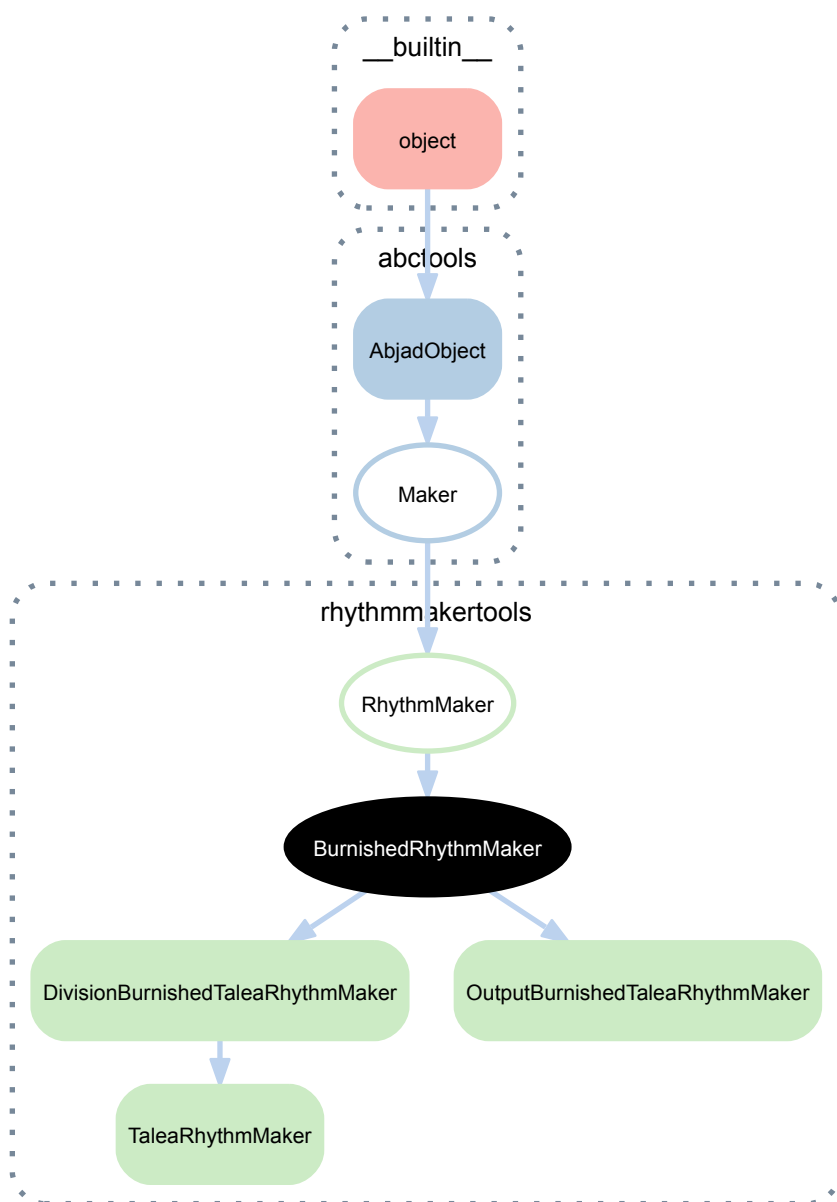
Returns list of rests.



## RHYTHMAKERTOOLS

### 23.1 Abstract classes

#### 23.1.1 `rhythmmakertools.BurnishedRhythmMaker`



```
class rhythmmakertools.BurnishedRhythmMaker (talea=None,      talea_denominator=None,
                                              prolation_addenda=None,    lefts=None,
                                              middles=None,              rights=None,
                                              left_lengths=None,    right_lengths=None,
                                              secondary_divisions=None,
                                              talea_helper=None,      prola-
                                              tion_addenda_helper=None,
                                              lefts_helper=None,      mid-
                                              dles_helper=None,    rights_helper=None,
                                              left_lengths_helper=None,
                                              right_lengths_helper=None,    sec-
                                              ondary_divisions_helper=None,
                                              beam_each_cell=False,
                                              beam_cells_together=False,    de-
                                              crease_durations_monotonically=True,
                                              tie_split_notes=False, tie_rests=False)
```

Abstract base class for rhythm-makers that burnish some or all of the output cells they produce.

‘Burnishing’ means to forcibly cast the first or last (or both first and last) elements of a output cell to be either a note or rest.

‘Division-burnishing’ rhythm-makers burnish every output cell they produce.

‘Output-burnishing’ rhythm-makers burnish only the first and last output cells they produce and leave interior output cells unchanged.

## Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`BurnishedRhythmMaker.storage_format`

Burnished rhythm-maker storage format.

Returns string.

## Methods

`(RhythmMaker).new (*kwargs)`

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = maker.new(decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

`BurnishedRhythmMaker.reverse()`

Reverse burnished rhythm-maker.

Defined equal to a copy of rhythm-maker with all the following lists reversed:

```
new.talea
new.prolation_addenda
new.lefts
new.middles
new.rights
new.left_lengths
new.right_lengths
new.secondary_divisions
```

Returns newly constructed rhythm-maker.

## Special methods

`BurnishedRhythmMaker.__call__(divisions, seeds=None)`

Call burnished rhythm-maker on *divisions*.

Returns either list of tuplets or else list of note-lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

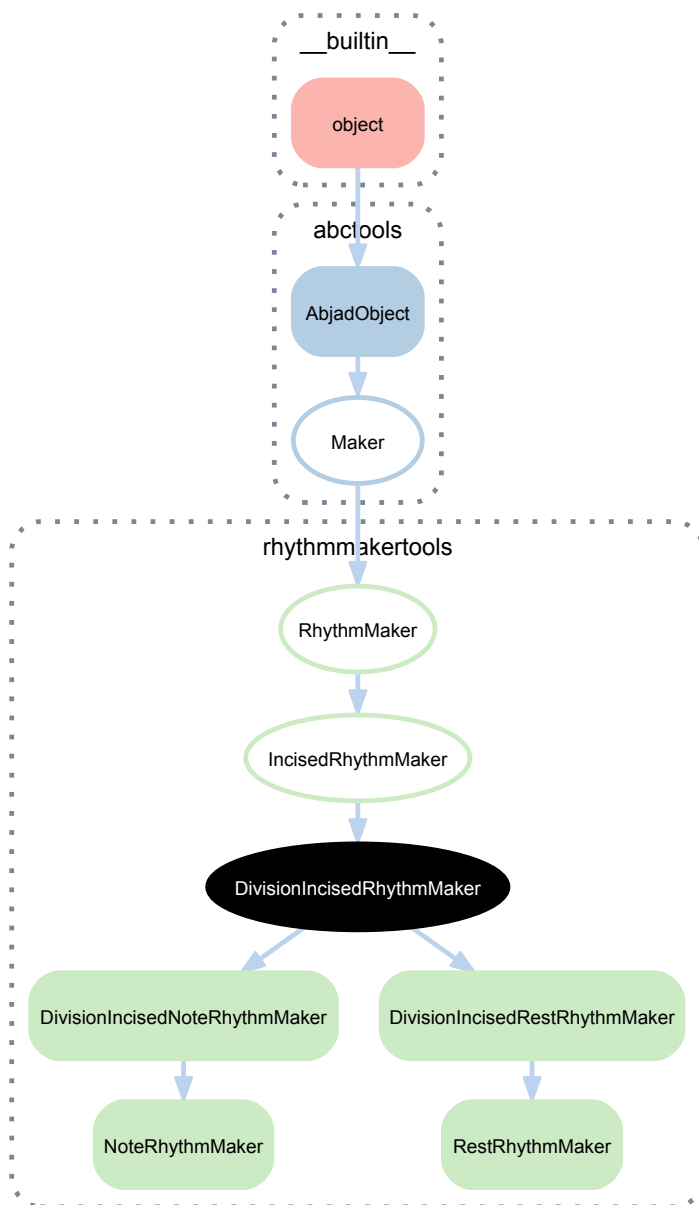
Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

## 23.1.2 rhythmtools.DivisionIncisedRhythmMaker





```

class rhythmmakertools.DivisionIncisedRhythmMaker (prefix_talea=None,      pre-
                                                    fix_lengths=None,      suf-
                                                    fix_talea=None,      suf-
                                                    fix_lengths=None,
                                                    talea_denominator=None,
                                                    body_ratio=None,      prola-
                                                    tion_addenda=None,      sec-
                                                    ondary_divisions=None,      pre-
                                                    fix_talea_helper=None,      pre-
                                                    fix_lengths_helper=None,      suf-
                                                    fix_talea_helper=None,      suf-
                                                    fix_lengths_helper=None,      prola-
                                                    tion_addenda_helper=None,      sec-
                                                    ondary_divisions_helper=None,
                                                    de-
                                                    crease_durations_monotonically=True,
                                                    tie_rests=False,      forbid-
                                                    den_written_duration=None,
                                                    beam_each_cell=False,
                                                    beam_cells_together=False)

```

Abstract base class for rhythm-makers that incise every output cell they produce.

## Bases

- `rhythmmakertools.IncisedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Maker) **.storage\_format**  
 Storage format of maker.  
 Returns string.

## Methods

(RhythmMaker) **.new** (*\*\*kwargs*)  
 Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```

>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = maker.new(decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)

```

```

>>> staff = Staff(
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)

```

Returns new rhythm-maker.

`(IncisedRhythmMaker).reverse()`  
Reverse incised rhythm-maker.  
Returns newly constructed rhythm-maker.

### Special methods

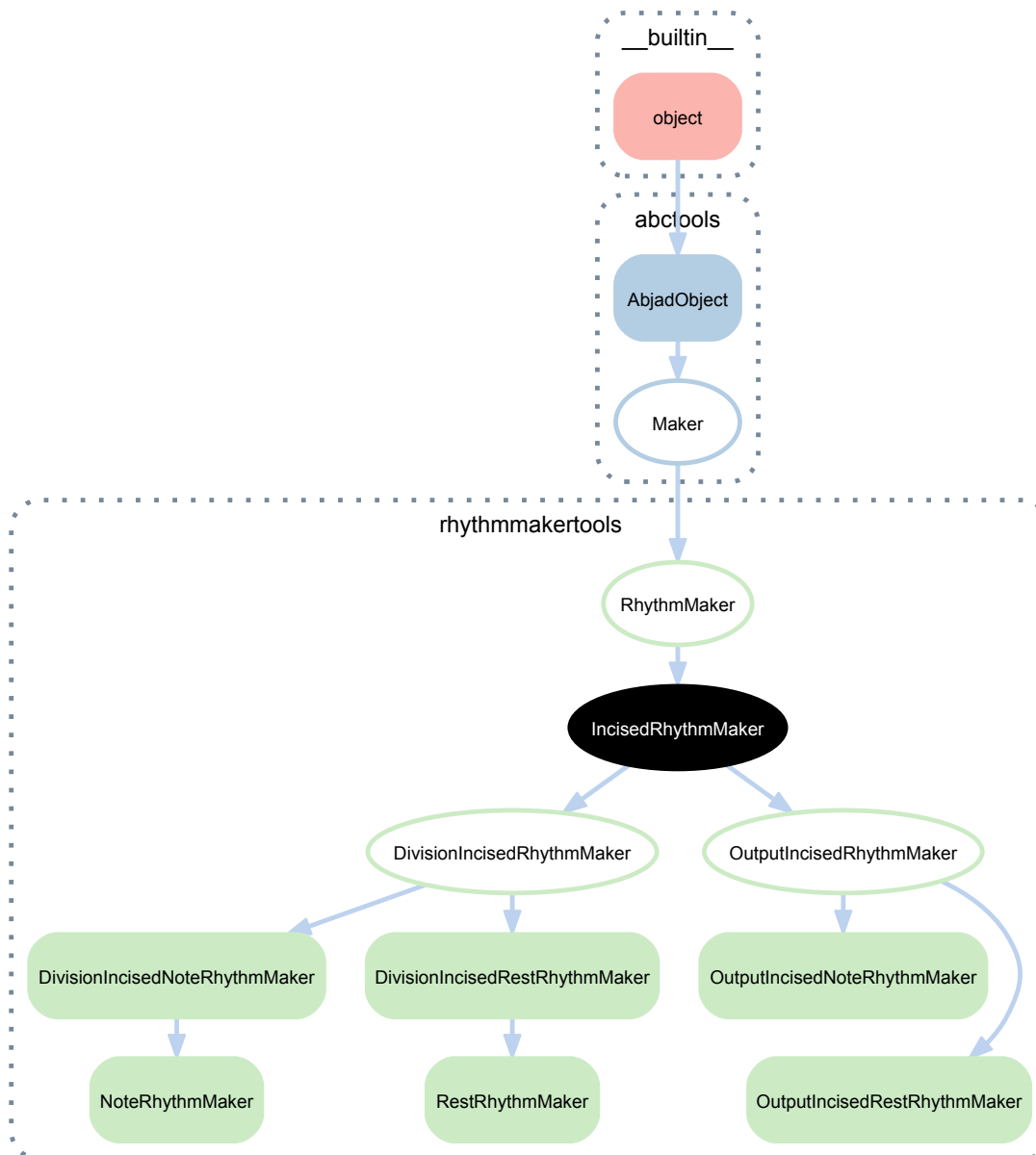
`(IncisedRhythmMaker).__call__(divisions, seeds=None)`  
Call incised rhythm-maker on *divisions*.  
Returns list of tuplelets or return list of leaf lists.

`(RhythmMaker).__eq__(expr)`  
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(RhythmMaker).__repr__()`  
Rhythm-maker interpreter representation.  
Returns string.

### 23.1.3 rhythm makertools.IncisedRhythmMaker



```

class rhythm makertools.IncisedRhythmMaker (prefix_talea=None,      prefix_lengths=None,
                                              suffix_talea=None,      suffix_lengths=None,
                                              talea_denominator=None,  body_ratio=None,
                                              prolation_addenda=None,  secondary_divisions=None,
                                              fix_talea_helper=None,    pre-
                                              fix_lengths_helper=None,  suf-
                                              fix_talea_helper=None,    suf-
                                              fix_lengths_helper=None,  prola-
                                              tion_addenda_helper=None, secondary_divisions_helper=None,
                                              decrease_durations_monotonically=True,
                                              tie_rests=False,         forbid-
                                              den_written_duration=None,
                                              beam_each_cell=False,
                                              beam_cells_together=False)

```

Abstract base class for rhythm-makers that incise some or all of the output cells they produce.

Rhythm makers can incise the edge of every output cell.

Or rhythm-makers can incise only the start of the first output cell and the end of the last output cell.

## Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Maker) **.storage\_format**

Storage format of maker.

Returns string.

## Methods

(RhythmMaker) **.new** (*\*\*kwargs*)

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = maker.new(decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

IncisedRhythmMaker **.reverse** ()

Reverse incised rhythm-maker.

Returns newly constructed rhythm-maker.

## Special methods

IncisedRhythmMaker **.\_\_call\_\_** (*divisions*, *seeds=None*)

Call incised rhythm-maker on *divisions*.

Returns list of tuplets or return list of leaf lists.

(RhythmMaker) **.\_\_eq\_\_** (*expr*)

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

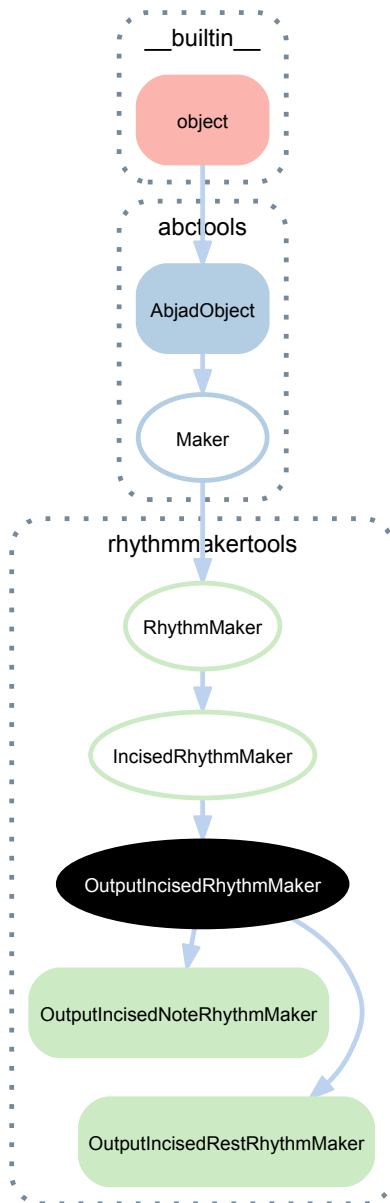
(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(RhythmMaker).__repr__()`  
 Rhythm-maker interpreter representation.  
 Returns string.

### 23.1.4 `rhythmmakertools.OutputIncisedRhythmMaker`



```
class rhythmtools.OutputIncisedRhythmMaker (prefix_talea=None,          pre-
                                             fix_lengths=None,          suf-
                                             fix_talea=None, suffix_lengths=None,
                                             talea_denominator=None,
                                             body_ratio=None,          prola-
                                             tion_addenda=None,          sec-
                                             ondary_divisions=None,      pre-
                                             fix_talea_helper=None,      pre-
                                             fix_lengths_helper=None,      suf-
                                             fix_talea_helper=None,      suf-
                                             fix_lengths_helper=None,      prola-
                                             tion_addenda_helper=None,      sec-
                                             ondary_divisions_helper=None, de-
                                             crease_durations_monotonically=True,
                                             tie_rests=False,          forbid-
                                             den_written_duration=None,
                                             beam_each_cell=False,
                                             beam_cells_together=False)
```

Abstract base class for rhythm-makers that incise only the first and last output cells they produce.

## Bases

- `rhythmtools.IncisedRhythmMaker`
- `rhythmtools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Maker) **.storage\_format**

Storage format of maker.

Returns string.

## Methods

(RhythmMaker) **.new** (*\*kwargs*)

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmtools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = maker.new(decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

(IncisedRhythmMaker) **.reverse** ()

Reverse incised rhythm-maker.

Returns newly constructed rhythm-maker.

## Special methods

`(IncisedRhythmMaker).__call__(divisions, seeds=None)`

Call incised rhythm-maker on *divisions*.

Returns list of tuplets or return list of leaf lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

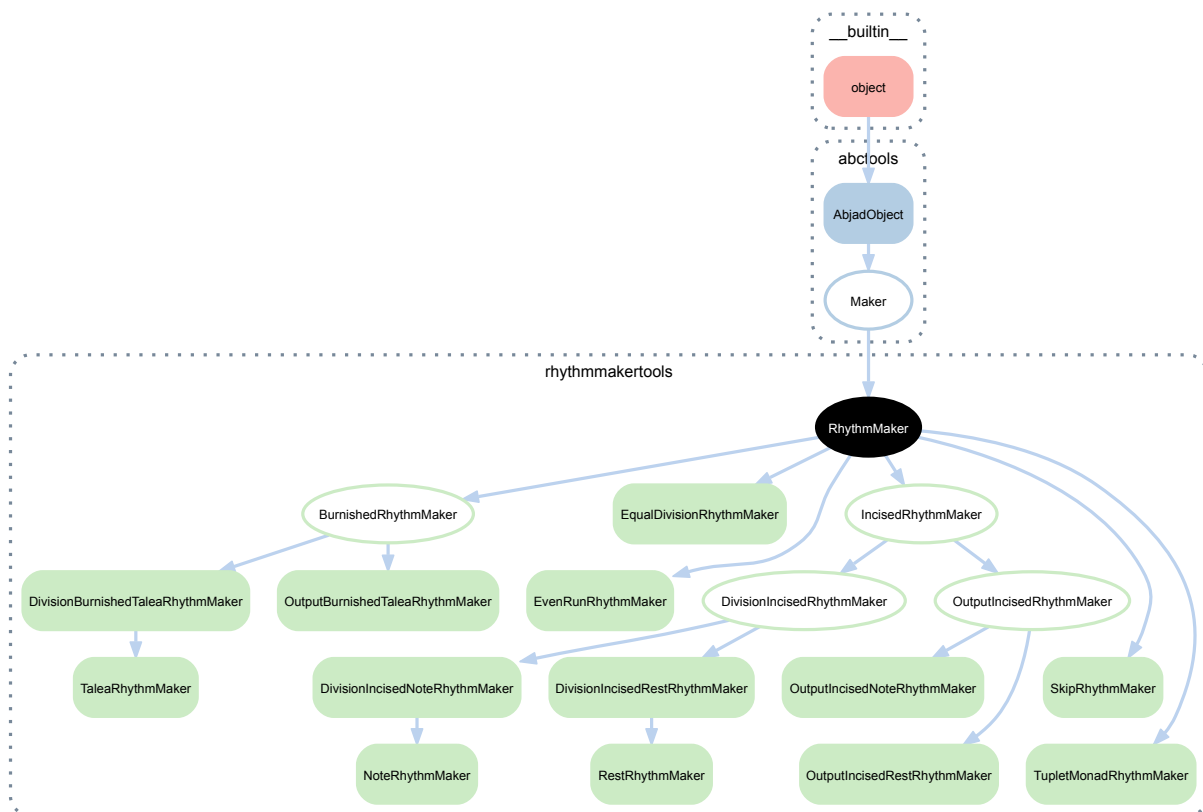
Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

## 23.1.5 rhythm makertools.RhythmMaker



**class** `rhythm makertools.RhythmMaker` (*forbidden\_written\_duration=None*,  
*beam\_each\_cell=True, beam\_cells\_together=False*)  
 Rhythm maker abstract base class.

## Bases

- `abctools.Maker`

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Maker).storage_format`

Storage format of maker.

Returns string.

## Methods

`RhythmMaker.new(**kwargs)`

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = maker.new(decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

`RhythmMaker.reverse()`

Reverse rhythm-maker.

---

**Note:** method is provisional.

---

Defined equal to exact copy of rhythm-maker.

This is the fallback for child classes.

Directed rhythm-maker child classes should override this method.

Returns newly constructed rhythm-maker.

## Special methods

`RhythmMaker.__call__(divisions, seeds=None)`

Cast *divisions* into duration pairs. Reduce numerator and denominator relative to each other.

Change none *seeds* into empty list.

Returns duration pairs and seed list.

`RhythmMaker.__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

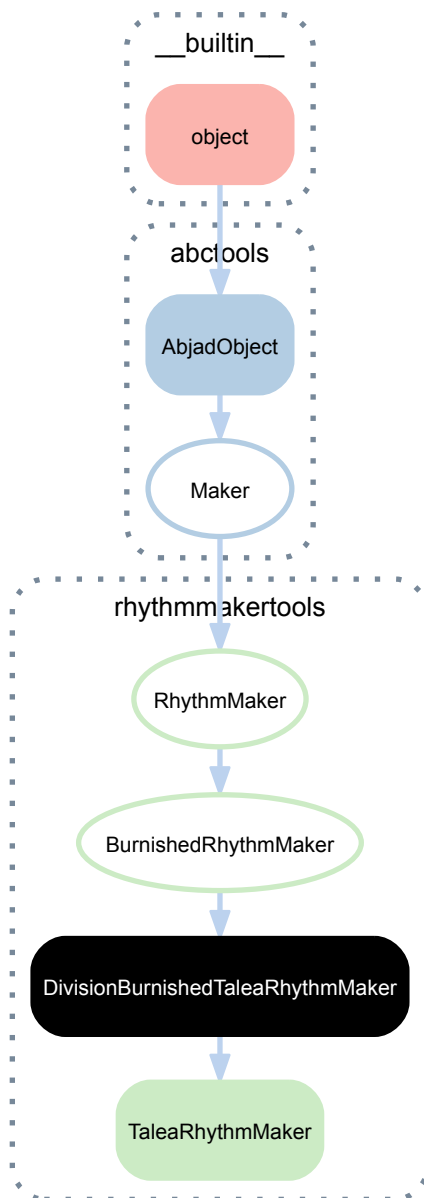
Returns boolean.



`RhythmMaker.__repr__()`  
 Rhythm-maker interpreter representation.  
 Returns string.

## 23.2 Concrete classes

### 23.2.1 `rhythmmakertools.DivisionBurnishedTaleaRhythmMaker`



```
class rhythmmakertools.DivisionBurnishedTaleaRhythmMaker (talea=None,
                                                         talea_denominator=None,
                                                         prola-
                                                         tion_addenda=None,
                                                         lefts=None,      mid-
                                                         dles=None,
                                                         rights=None,
                                                         left_lengths=None,
                                                         right_lengths=None,
                                                         sec-
                                                         ondary_divisions=None,
                                                         talea_helper=None,
                                                         prola-
                                                         tion_addenda_helper=None,
                                                         lefts_helper=None, mid-
                                                         dles_helper=None,
                                                         rights_helper=None,
                                                         left_lengths_helper=None,
                                                         right_lengths_helper=None,
                                                         sec-
                                                         ondary_divisions_helper=None,
                                                         beam_each_cell=False,
                                                         beam_cells_together=False,
                                                         de-
                                                         crease_durations_monotonically=True,
                                                         tie_split_notes=False,
                                                         tie_rests=False)
```

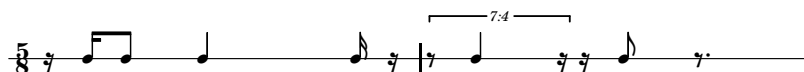
Division-burnished talea rhythm-maker:

```
>>> maker = rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
...     talea=[1, 1, 2, 4],
...     talea_denominator=16,
...     prolation_addenda=[0, 3],
...     lefts=[-1],
...     middles=[0],
...     rights=[-1],
...     left_lengths=[1],
...     right_lengths=[1],
...     secondary_divisions=[14])
```

Configure at instantiation and then call on any sequence of divisions:

```
>>> divisions = [(5, 8), (5, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.BurnishedRhythmMaker`

- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`DivisionBurnishedTaleaRhythmMaker.storage_format`

Division-burnished talea rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[1, 1, 2, 4],
    talea_denominator=16,
    prolation_addenda=[0, 3],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[14],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False
)
```

Returns string.

## Methods

`DivisionBurnishedTaleaRhythmMaker.new(**kwargs)`

Create new rhythm-maker with *kwargs*:

```
>>> print maker.storage_format
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[1, 1, 2, 4],
    talea_denominator=16,
    prolation_addenda=[0, 3],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[14],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False
)
```

```
>>> new_maker = maker.new(talea=[1, 1, 2])
```

```
>>> print new_maker.storage_format
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[1, 1, 2],
    talea_denominator=16,
    prolation_addenda=[0, 3],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
)
```

```

secondary_divisions=[14],
beam_each_cell=False,
beam_cells_together=False,
decrease_durations_monotonically=True,
tie_split_notes=False,
tie_rests=False
)

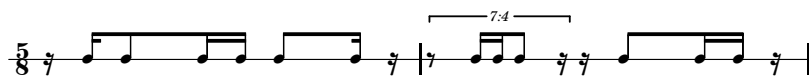
```

```

>>> divisions = [(5, 8), (5, 8)]
>>> music = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)

```

```
>>> show(staff)
```



Returns new division-burnished talea rhythm-maker.

`DivisionBurnishedTaleaRhythmMaker.reverse()`

Reverse division-burnished talea rhythm-maker.

Nonreversed output:

```

>>> print maker.storage_format
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[1, 1, 2, 4],
    talea_denominator=16,
    prolation_addenda=[0, 3],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[14],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False
)

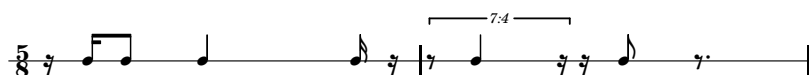
```

```

>>> divisions = [(5, 8), (5, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)

```

```
>>> show(staff)
```



Reversed output:

```
>>> reversed_maker = maker.reverse()
```

```

>>> print reversed_maker.storage_format
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(

```

```

talea=[4, 2, 1, 1],
talea_denominator=16,
prolation_addenda=[3, 0],
lefts=[-1],
middles=[0],
rights=[-1],
left_lengths=[1],
right_lengths=[1],
secondary_divisions=[14],
beam_each_cell=False,
beam_cells_together=False,
decrease_durations_monotonically=False,
tie_split_notes=False,
tie_rests=False
)

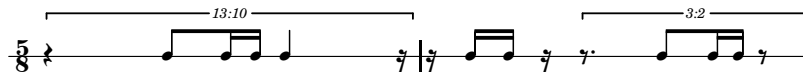
```

```

>>> divisions = [(5, 8), (5, 8)]
>>> music = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)

```

```
>>> show(staff)
```



Returns new division-burnished talea rhythm-maker.

## Special methods

(BurnishedRhythmMaker) **.\_\_call\_\_** (*divisions*, *seeds=None*)

Call burnished rhythm-maker on *divisions*.

Returns either list of tuplets or else list of note-lists.

(RhythmMaker) **.\_\_eq\_\_** (*expr*)

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

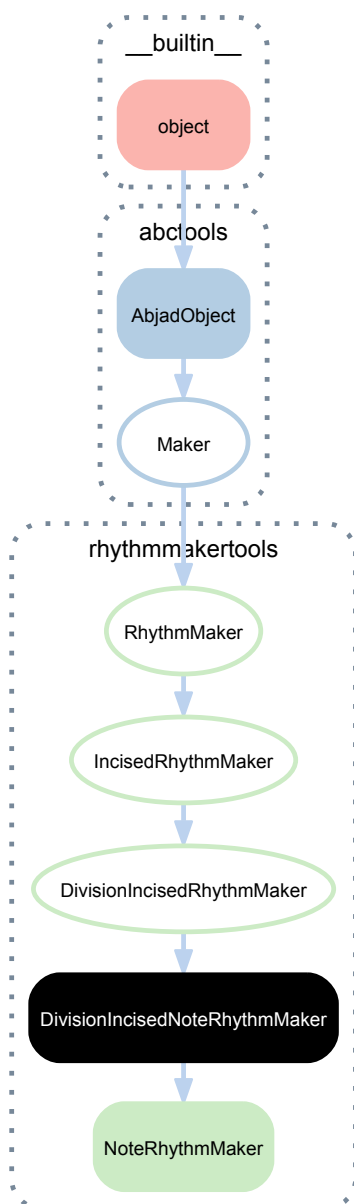
Returns boolean.

(RhythmMaker) **.\_\_repr\_\_** ()

Rhythm-maker interpreter representation.

Returns string.

### 23.2.2 `rhythmmakertools.DivisionIncisedNoteRhythmMaker`



```
class rhythmmakertools.DivisionIncisedNoteRhythmMaker (prefix_talea=None,    pre-
                                                         fix_lengths=None,    suf-
                                                         fix_talea=None,    suf-
                                                         fix_lengths=None,
                                                         talea_denominator=None,
                                                         body_ratio=None,    pro-
                                                         lation_addenda=None,
                                                         secondary_divisions=None,
                                                         prefix_talea_helper=None,
                                                         prefix_lengths_helper=None,
                                                         suffix_talea_helper=None,
                                                         suffix_lengths_helper=None,
                                                         prola-
                                                         tion_addenda_helper=None,
                                                         sec-
                                                         ondary_divisions_helper=None,
                                                         de-
                                                         crease_durations_monotonically=True,
                                                         tie_rests=False,    forbid-
                                                         den_written_duration=None,
                                                         beam_each_cell=False,
                                                         beam_cells_together=False)
```

Division-incised note rhythm-maker:

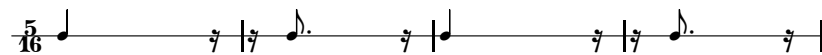
**Example 1.** Basic usage:

```
>>> maker = rhythmmakertools.DivisionIncisedNoteRhythmMaker (
...     prefix_talea=[-1],
...     prefix_lengths=[0, 1],
...     suffix_talea=[-1],
...     suffix_lengths=[1],
...     talea_denominator=16)
```

Configure at instantiation and then call on any sequence of divisions:

```
>>> divisions = 4 * [(5, 16)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips (
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr (
...     staff, leaves)
```

```
>>> show(staff)
```



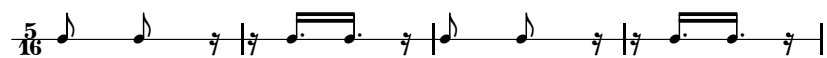
**Example 2.** Set *body\_ratio* to divide middle part proportionally:

```
>>> maker = rhythmmakertools.DivisionIncisedNoteRhythmMaker (
...     prefix_talea=[-1],
...     prefix_lengths=[0, 1],
...     suffix_talea=[-1],
...     suffix_lengths=[1],
...     talea_denominator=16,
...     body_ratio=(1, 1))
```

```
>>> divisions = 4 * [(5, 16)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips (
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
```

```
>>> measures = measuretools.replace_contents_of_measures_in_expr(  
...     staff, leaves)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.DivisionIncisedRhythmMaker`
- `rhythmmakertools.IncisedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`DivisionIncisedNoteRhythmMaker.storage_format`

Division-incised note rhythm-maker storage format:

```
>>> print maker.storage_format  
rhythmmakertools.DivisionIncisedNoteRhythmMaker(  
    prefix_talea=[-1],  
    prefix_lengths=[0, 1],  
    suffix_talea=[-1],  
    suffix_lengths=[1],  
    talea_denominator=16,  
    body_ratio=mathtools.Ratio(1, 1),  
    prolation_addenda=[],  
    secondary_divisions=[],  
    decrease_durations_monotonically=True,  
    tie_rests=False,  
    beam_each_cell=False,  
    beam_cells_together=False  
)
```

Returns string.

## Methods

`DivisionIncisedNoteRhythmMaker.new(**kwargs)`

Create new division-incised note rhythm-maker with *kwargs*:

```
>>> print maker.storage_format  
rhythmmakertools.DivisionIncisedNoteRhythmMaker(  
    prefix_talea=[-1],  
    prefix_lengths=[0, 1],  
    suffix_talea=[-1],  
    suffix_lengths=[1],  
    talea_denominator=16,  
    body_ratio=mathtools.Ratio(1, 1),  
    prolation_addenda=[],  
    secondary_divisions=[],  
    decrease_durations_monotonically=True,  
    tie_rests=False,  
    beam_each_cell=False,  
    beam_cells_together=False  
)
```

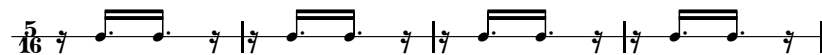


```
>>> new_maker = maker.new(prefix_lengths=[1])
```

```
>>> print new_maker.storage_format
rhythmmakertools.DivisionIncisedNoteRhythmMaker(
  prefix_talea=[-1],
  prefix_lengths=[1],
  suffix_talea=[-1],
  suffix_lengths=[1],
  talea_denominator=16,
  body_ratio=mathtools.Ratio(1, 1),
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new division-incised note rhythm-maker.

`DivisionIncisedNoteRhythmMaker.reverse()`

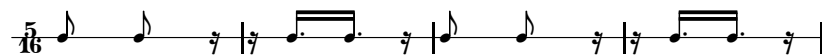
Reverse division-incised note rhythm-maker.

Nonreversed output:

```
>>> print maker.storage_format
rhythmmakertools.DivisionIncisedNoteRhythmMaker(
  prefix_talea=[-1],
  prefix_lengths=[0, 1],
  suffix_talea=[-1],
  suffix_lengths=[1],
  talea_denominator=16,
  body_ratio=mathtools.Ratio(1, 1),
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



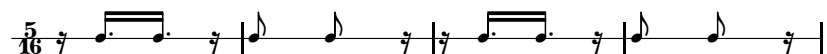
Reversed output:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.DivisionIncisedNoteRhythmMaker(
  prefix_talea=[-1],
  prefix_lengths=[1, 0],
  suffix_talea=[-1],
  suffix_lengths=[1],
  talea_denominator=16,
  body_ratio=mathtools.Ratio(1, 1),
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=False,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...     divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns division-incised note rhythm-maker.

## Special methods

(IncisedRhythmMaker).**\_\_call\_\_**(*divisions*, *seeds=None*)

Call incised rhythm-maker on *divisions*.

Returns list of tuplets or return list of leaf lists.

(RhythmMaker).**\_\_eq\_\_**(*expr*)

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

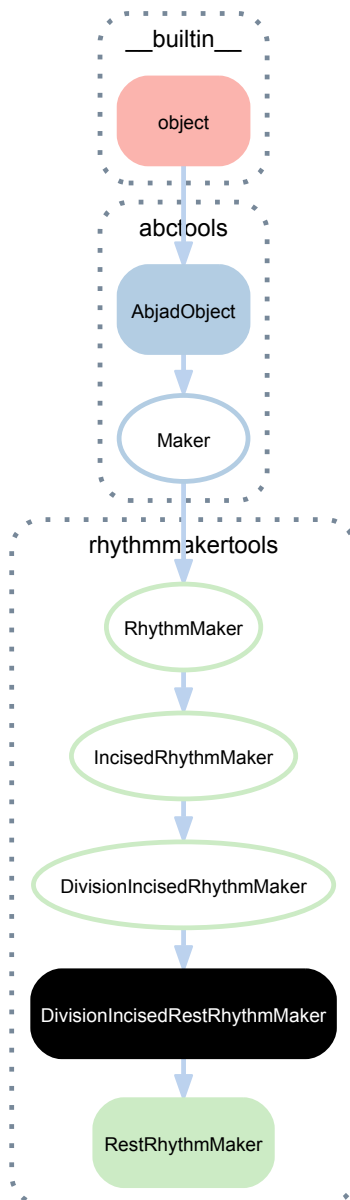
Returns boolean.

(RhythmMaker).**\_\_repr\_\_**()

Rhythm-maker interpreter representation.

Returns string.

### 23.2.3 `rhythmmakertools.DivisionIncisedRestRhythmMaker`



```
class rhythmmakertools.DivisionIncisedRestRhythmMaker (prefix_talea=None,    pre-
                                                         fix_lengths=None,    suf-
                                                         fix_talea=None,    suf-
                                                         fix_lengths=None,
                                                         talea_denominator=None,
                                                         body_ratio=None,    pro-
                                                         lation_addenda=None,
                                                         secondary_divisions=None,
                                                         prefix_talea_helper=None,
                                                         prefix_lengths_helper=None,
                                                         suffix_talea_helper=None,
                                                         suffix_lengths_helper=None,
                                                         prola-
                                                         tion_addenda_helper=None,
                                                         sec-
                                                         ondary_divisions_helper=None,
                                                         de-
                                                         crease_durations_monotonically=True,
                                                         tie_rests=False,    forbid-
                                                         den_written_duration=None,
                                                         beam_each_cell=False,
                                                         beam_cells_together=False)
```

Division-incised rest rhythm-maker:

```
>>> maker = rhythmmakertools.DivisionIncisedRestRhythmMaker (
...     prefix_talea=[1],
...     prefix_lengths=[1, 2, 3, 4],
...     suffix_talea=[1],
...     suffix_lengths=[1],
...     talea_denominator=32)
```

Configure at instantiation and then call on any sequence of divisions:

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips (
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr (
...     staff, leaves)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.DivisionIncisedRhythmMaker`
- `rhythmmakertools.IncisedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`DivisionIncisedRestRhythmMaker.storage_format`

Division-incised rest rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.DivisionIncisedRestRhythmMaker(
  prefix_talea=[1],
  prefix_lengths=[1, 2, 3, 4],
  suffix_talea=[1],
  suffix_lengths=[1],
  talea_denominator=32,
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

Returns string.

## Methods

`DivisionIncisedRestRhythmMaker.new(**kwargs)`

Create new division-incised rest rhythm-maker with *kwargs*:

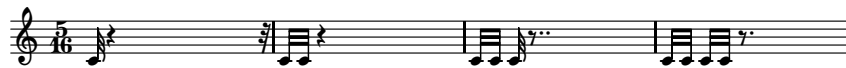
```
>>> print maker.storage_format
rhythmmakertools.DivisionIncisedRestRhythmMaker(
  prefix_talea=[1],
  prefix_lengths=[1, 2, 3, 4],
  suffix_talea=[1],
  suffix_lengths=[1],
  talea_denominator=32,
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

```
>>> new_maker = maker.new(suffix_lengths=[0])
```

```
>>> print new_maker.storage_format
rhythmmakertools.DivisionIncisedRestRhythmMaker(
  prefix_talea=[1],
  prefix_lengths=[1, 2, 3, 4],
  suffix_talea=[1],
  suffix_lengths=[0],
  talea_denominator=32,
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new division-incised rest rhythm-maker.

`DivisionIncisedRestRhythmMaker.reverse()`

Reverse division-incised rest rhythm-maker:

```
>>> print maker.storage_format
rhythmmakertools.DivisionIncisedRestRhythmMaker(
    prefix_talea=[1],
    prefix_lengths=[1, 2, 3, 4],
    suffix_talea=[1],
    suffix_lengths=[1],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
)
```

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.DivisionIncisedRestRhythmMaker(
    prefix_talea=[1],
    prefix_lengths=[4, 3, 2, 1],
    suffix_talea=[1],
    suffix_lengths=[1],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=False,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
)
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new division-incised rest rhythm-maker.

## Special methods

`(IncisedRhythmMaker).__call__(divisions, seeds=None)`

Call incised rhythm-maker on *divisions*.

Returns list of tuplets or return list of leaf lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

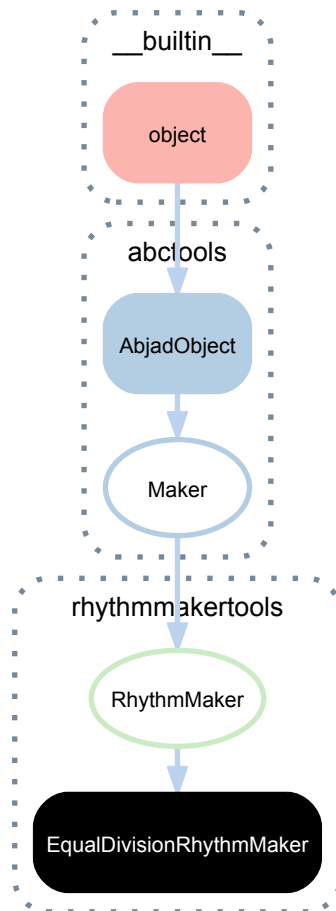
(AbjadObject).**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(RhythmMaker).**\_\_repr\_\_**()  
Rhythm-maker interpreter representation.

Returns string.

### 23.2.4 `rhythmmakertools.EqualDivisionRhythmMaker`



```
class rhythmmakertools.EqualDivisionRhythmMaker (leaf_count=None,
                                                    is_diminution=True,
                                                    beam_each_cell=True,
                                                    beam_cells_together=False)
```

Equal division rhythm-maker:

```
>>> maker = rhythmmakertools.EqualDivisionRhythmMaker(leaf_count=4)
```

Configure at initialization and then call on any series of divisions:

```
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> tuplet_lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(tuplet_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`EqualDivisionRhythmMaker.is_diminution`

True when output tuplets should be diminished.

False when output tuplets should be augmented:

```
>>> maker.is_diminution
True
```

Returns boolean.

`EqualDivisionRhythmMaker.leaf_count`

Number of leaves per division:

```
>>> maker.leaf_count
4
```

Returns positive integer.

`EqualDivisionRhythmMaker.storage_format`

Equal-division rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.EqualDivisionRhythmMaker(
  leaf_count=4,
  is_diminution=True,
  beam_each_cell=True,
  beam_cells_together=False
)
```

Returns string.

## Methods

`EqualDivisionRhythmMaker.new(**kwargs)`

Create new equal-division rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(is_diminution=False)
```

```
>>> print new_maker.storage_format
rhythmmakertools.EqualDivisionRhythmMaker(
  leaf_count=4,
  is_diminution=False,
  beam_each_cell=True,
  beam_cells_together=False
)
```



```
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> tuplet_lists = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(tuplet_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Returns new equal-division rhythm-maker.

`EqualDivisionRhythmMaker.reverse()`

Reverse equal-division rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.EqualDivisionRhythmMaker(
    leaf_count=4,
    is_diminution=True,
    beam_each_cell=True,
    beam_cells_together=False
)
```

```
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> tuplet_lists = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(tuplet_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Defined equal to copy of maker.

Returns new equal-division rhythm-maker.

## Special methods

`EqualDivisionRhythmMaker.__call__(divisions, seeds=None)`

Call equal-division rhythm-maker on *divisions*.

Returns list of tuplet lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

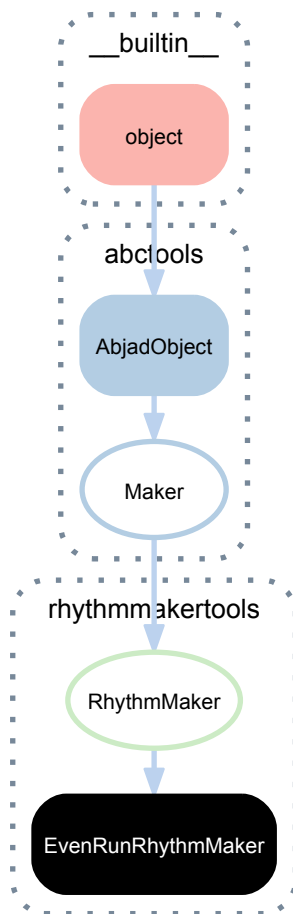
Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

### 23.2.5 rhythm makertools.EvenRunRhythmMaker



**class** `rhythm makertools.EvenRunRhythmMaker` (*denominator\_multiplier\_exponent=0*,  
*beam\_each\_cell=True*,  
*beam\_cells\_together=False*)

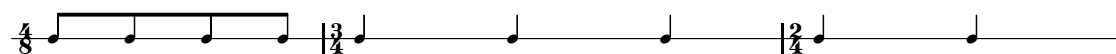
Even run rhythm-maker.

**Example 1.** Make even run of notes each equal in duration to  $1/d$  with  $d$  equal to the denominator of each division on which the rhythm-maker is called:

```
>>> maker = rhythm makertools.EvenRunRhythmMaker()
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```

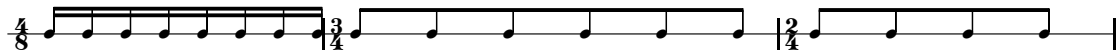


**Example 2.** Make even run of notes each equal in duration to  $1/(2*d)$  with  $d$  equal to the denominator of each division on which the rhythm-maker is called:

```
>>> maker = rhythmmakertools.EvenRunRhythmMaker(1)
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Output a list of lists of depth-2 note-bearing containers.

Even-run rhythm-maker doesn't yet work with non-power-of-two divisions.

Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`EvenRunRhythmMaker.denominator_multiplier_exponent`  
Denominator multiplier exponent provided at initialization.

```
>>> maker.denominator_multiplier_exponent
1
```

Returns nonnegative integer.

`EvenRunRhythmMaker.storage_format`  
Even-run rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.EvenRunRhythmMaker(
    denominator_multiplier_exponent=1,
    beam_each_cell=True,
    beam_cells_together=False
)
```

Returns string.

## Methods

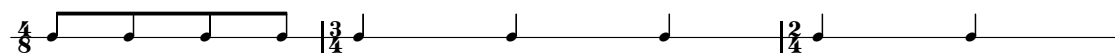
`EvenRunRhythmMaker.new(**kwargs)`  
Create new even-run rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(denominator_multiplier_exponent=0)
```

```
>>> print new_maker.storage_format
rhythmmakertools.EvenRunRhythmMaker(
    denominator_multiplier_exponent=0,
    beam_each_cell=True,
    beam_cells_together=False
)
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Returns new even-run rhythm-maker.

`EvenRunRhythmMaker.reverse()`

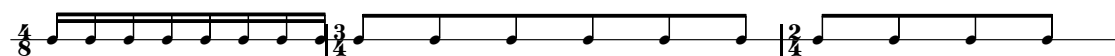
Reverse even-run rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.EvenRunRhythmMaker(
    denominator_multiplier_exponent=1,
    beam_each_cell=True,
    beam_cells_together=False
)
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Defined equal to copy of even-run rhythm-maker.

Returns new even-run rhythm-maker.

## Special methods

`EvenRunRhythmMaker.__call__(divisions, seeds=None)`

Call even-run rhythm-maker on *divisions*.

Returns list of container lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

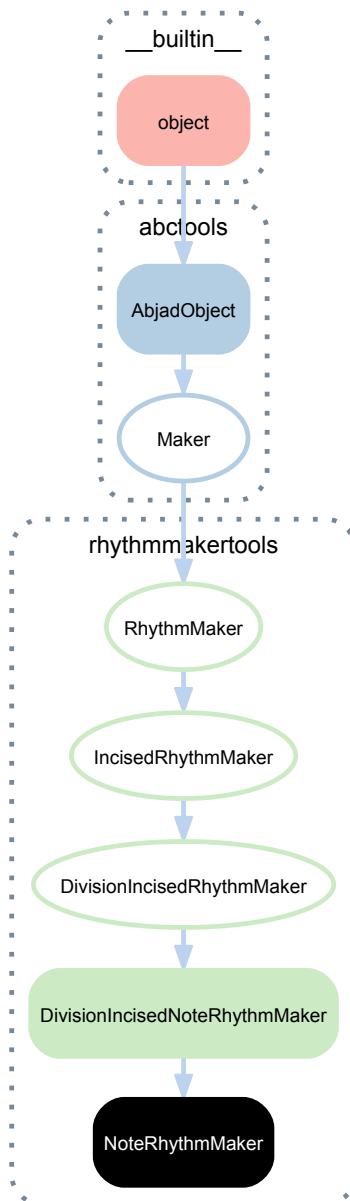
Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

### 23.2.6 `rhythmmakertools.NoteRhythmMaker`



**class** `rhythmmakertools.NoteRhythmMaker` (*decrease\_durations\_monotonically=True, forbid-*  
*den\_written\_duration=None, tie\_rests=False*)

Note rhythm-maker:

**Example 1:**

```

>>> maker = rhythmmakertools.NoteRhythmMaker()

>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(

```

```
...     divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```

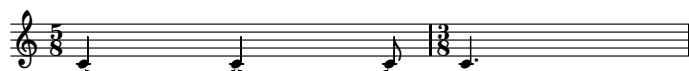


**Example 2.** Forbid half notes:

```
>>> maker = rhythmtools.NoteRhythmMaker(
...     forbidden_written_duration=Duration(1, 2))
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmtools.DivisionIncisedNoteRhythmMaker`
- `rhythmtools.DivisionIncisedRhythmMaker`
- `rhythmtools.IncisedRhythmMaker`
- `rhythmtools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`NoteRhythmMaker`.**storage\_format**

Note rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmtools.NoteRhythmMaker(
    decrease_durations_monotonically=True,
    forbidden_written_duration=durationtools.Duration(1, 2),
    tie_rests=False
)
```

Returns string.

## Methods

`NoteRhythmMaker.new(**kwargs)`

Create new note rhythm-maker:

```
>>> new_maker = maker.new(decrease_durations_monotonically=False)
```

```
>>> print new_maker.storage_format
rhythmmakertools.NoteRhythmMaker(
  decrease_durations_monotonically=False,
  forbidden_written_duration=durationtools.Duration(1, 2),
  tie_rests=False
)
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new note rhythm-maker.

`NoteRhythmMaker.reverse()`

Reverse note rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.NoteRhythmMaker(
  decrease_durations_monotonically=False,
  forbidden_written_duration=durationtools.Duration(1, 2),
  tie_rests=False
)
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new note rhythm-maker.

## Special methods

`(IncisedRhythmMaker).__call__(divisions, seeds=None)`

Call incised rhythm-maker on *divisions*.

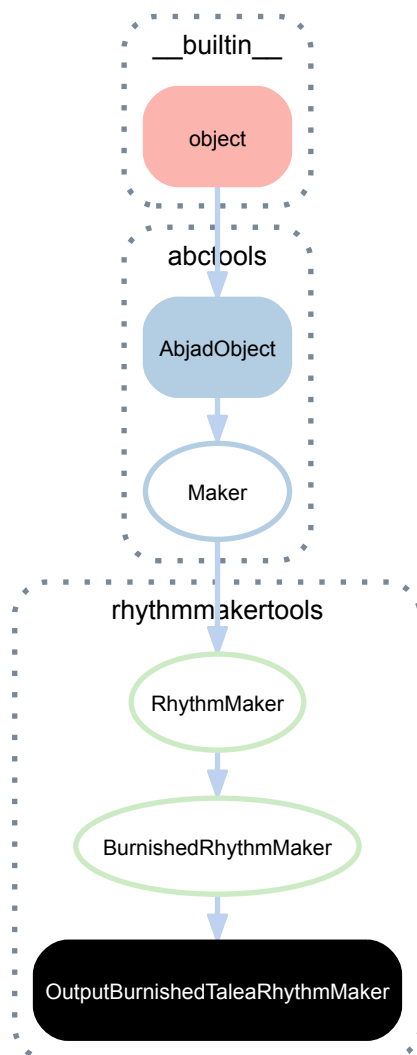
Returns list of tuplets or return list of leaf lists.

(RhythmMaker).**\_\_eq\_\_**(*expr*)  
 True when *expr* is same type with the equal public nonhelper properties. Otherwise false.  
 Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

NoteRhythmMaker.**\_\_repr\_\_**()  
 Note rhythm-maker interpreter representation.  
 Returns string.

### 23.2.7 rhythm makertools.OutputBurnishedTaleaRhythmMaker





```
class rhythmmakertools.OutputBurnishedTaleaRhythmMaker (talea=None,
                                                         talea_denominator=None,
                                                         prolation_addenda=None,
                                                         lefts=None,      mid-
                                                         dles=None,  rights=None,
                                                         left_lengths=None,
                                                         right_lengths=None,  sec-
                                                         ondary_divisions=None,
                                                         talea_helper=None, prola-
                                                         tion_addenda_helper=None,
                                                         lefts_helper=None,
                                                         middles_helper=None,
                                                         rights_helper=None,
                                                         left_lengths_helper=None,
                                                         right_lengths_helper=None,
                                                         sec-
                                                         ondary_divisions_helper=None,
                                                         beam_each_cell=False,
                                                         beam_cells_together=False,
                                                         de-
                                                         crease_durations_monotonically=True,
                                                         tie_split_notes=False,
                                                         tie_rests=False)
```

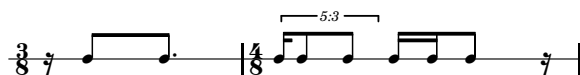
Output-burnished talea rhythm-maker:

```
>>> maker = rhythmmakertools.OutputBurnishedTaleaRhythmMaker(
...     talea=[1, 2, 3],
...     talea_denominator=16,
...     prolation_addenda=[0, 2],
...     lefts=[-1],
...     middles=[0],
...     rights=[-1],
...     left_lengths=[1],
...     right_lengths=[1],
...     secondary_divisions=[9],
...     beam_each_cell=True)
```

Configure at initialization and then call on any list of divisions:

```
>>> divisions = [(3, 8), (4, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.BurnishedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`

- `__builtin__.object`

## Read-only properties

`OutputBurnishedTaleaRhythmMaker.storage_format`

Output-burnished talea rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.OutputBurnishedTaleaRhythmMaker(
  talea=[1, 2, 3],
  talea_denominator=16,
  prolation_addenda=[0, 2],
  lefts=[-1],
  middles=[0],
  rights=[-1],
  left_lengths=[1],
  right_lengths=[1],
  secondary_divisions=[9],
  beam_each_cell=True,
  beam_cells_together=False,
  decrease_durations_monotonically=True,
  tie_split_notes=False,
  tie_rests=False
)
```

Returns string.

## Methods

`OutputBurnishedTaleaRhythmMaker.new(**kwargs)`

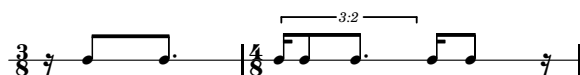
Create new output-burnished talea rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(secondary_divisions=[10])
```

```
>>> print new_maker.storage_format
rhythmmakertools.OutputBurnishedTaleaRhythmMaker(
  talea=[1, 2, 3],
  talea_denominator=16,
  prolation_addenda=[0, 2],
  lefts=[-1],
  middles=[0],
  rights=[-1],
  left_lengths=[1],
  right_lengths=[1],
  secondary_divisions=[10],
  beam_each_cell=True,
  beam_cells_together=False,
  decrease_durations_monotonically=True,
  tie_split_notes=False,
  tie_rests=False
)
```

```
>>> divisions = [(3, 8), (4, 8)]
>>> music = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Returns new output-burnished talea rhythm-maker.

`OutputBurnishedTaleaRhythmMaker.reverse()`  
Reverse output-burnished talea rhythm-maker:

```
>>> reversed_maker = maker.reverse()

>>> print reversed_maker.storage_format
rhythmmakertools.OutputBurnishedTaleaRhythmMaker(
  talea=[3, 2, 1],
  talea_denominator=16,
  prolation_addenda=[2, 0],
  lefts=[-1],
  middles=[0],
  rights=[-1],
  left_lengths=[1],
  right_lengths=[1],
  secondary_divisions=[9],
  beam_each_cell=True,
  beam_cells_together=False,
  decrease_durations_monotonically=False,
  tie_split_notes=False,
  tie_rests=False
)

>>> divisions = [(3, 8), (4, 8)]
>>> music = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)

>>> show(staff)
```



Returns new output-burnished talea rhythm-maker.

## Special methods

`(BurnishedRhythmMaker).__call__(divisions, seeds=None)`

Call burnished rhythm-maker on *divisions*.

Returns either list of tuplets or else list of note-lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

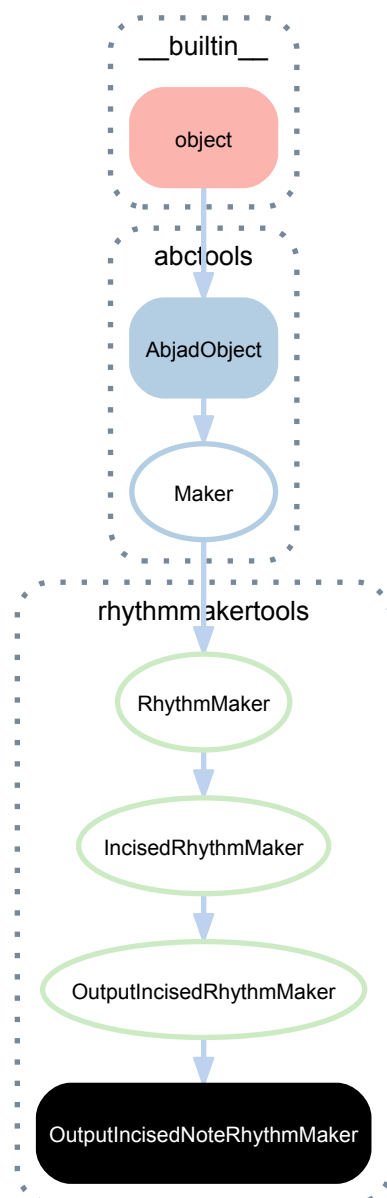
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

23.2.8 `rhythmmakertools.OutputIncisedNoteRhythmMaker`

```
class rhythmmakertools.OutputIncisedNoteRhythmMaker (prefix_talea=None,      pre-
                                                    fix_lengths=None,      suf-
                                                    fix_talea=None,      suf-
                                                    fix_lengths=None,
                                                    talea_denominator=None,
                                                    body_ratio=None,      prola-
                                                    tion_addenda=None,      sec-
                                                    ondary_divisions=None,
                                                    prefix_talea_helper=None,
                                                    prefix_lengths_helper=None,
                                                    suffix_talea_helper=None, suf-
                                                    fix_lengths_helper=None, pro-
                                                    lation_addenda_helper=None,
                                                    sec-
                                                    ondary_divisions_helper=None,
                                                    de-
                                                    crease_durations_monotonically=True,
                                                    tie_rests=False,      forbid-
                                                    den_written_duration=None,
                                                    beam_each_cell=False,
                                                    beam_cells_together=False)
```

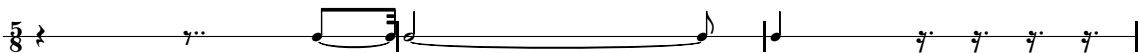
Output-incised note rhythm-maker:

```
>>> maker = rhythmmakertools.OutputIncisedNoteRhythmMaker(
...     prefix_talea=[-8, -7],
...     prefix_lengths=[2],
...     suffix_talea=[-3],
...     suffix_lengths=[4],
...     talea_denominator=32)
```

Configure at initialization and then call on arbitrary divisions:

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.OutputIncisedRhythmMaker`
- `rhythmmakertools.IncisedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`OutputIncisedNoteRhythmMaker.storage_format`

Output-incised note rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.OutputIncisedNoteRhythmMaker(
  prefix_talea=[-8, -7],
  prefix_lengths=[2],
  suffix_talea=[-3],
  suffix_lengths=[4],
  talea_denominator=32,
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

Returns string.

## Methods

`OutputIncisedNoteRhythmMaker.new(**kwargs)`

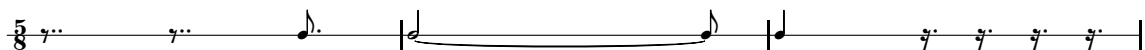
Create new output-incised note rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(prefix_talea=[-7])
```

```
>>> print new_maker.storage_format
rhythmmakertools.OutputIncisedNoteRhythmMaker(
  prefix_talea=[-7],
  prefix_lengths=[2],
  suffix_talea=[-3],
  suffix_lengths=[4],
  talea_denominator=32,
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new output-incised note rhythm-maker.

`OutputIncisedNoteRhythmMaker.reverse()`

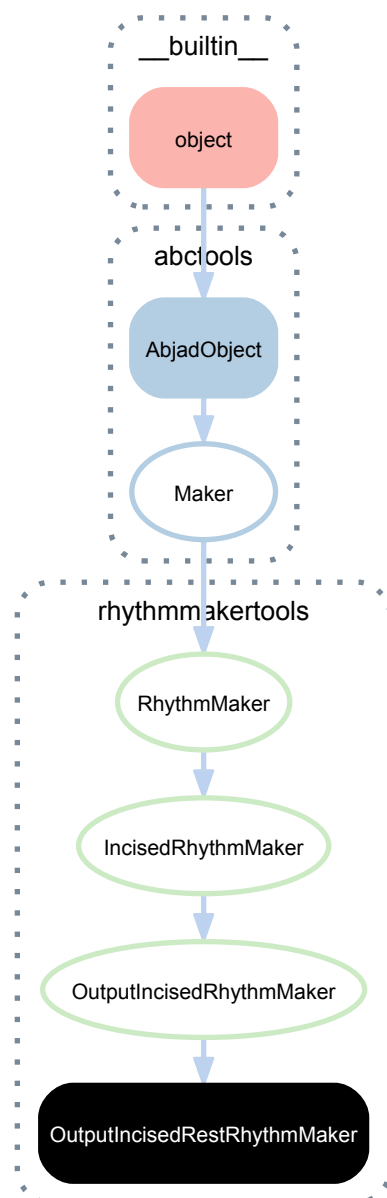
Reverse output-incised note rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.OutputIncisedNoteRhythmMaker(
  prefix_talea=[-7, -8],
```



## 23.2.9 rhythmtools.OutputIncisedRestRhythmMaker





```
class rhythmtools.OutputIncisedRestRhythmMaker (prefix_talea=None, pre-
fix_lengths=None, suf-
fix_talea=None, suf-
fix_lengths=None,
talea_denominator=None,
body_ratio=None, prola-
tion_addenda=None, sec-
ondary_divisions=None,
prefix_talea_helper=None,
prefix_lengths_helper=None,
suffix_talea_helper=None, suf-
fix_lengths_helper=None, pro-
lation_addenda_helper=None,
sec-
ondary_divisions_helper=None,
de-
crease_durations_monotonically=True,
tie_rests=False, forbid-
den_written_duration=None,
beam_each_cell=False,
beam_cells_together=False)
```

Output-incised rest rhythm-maker:

```
>>> maker = rhythmmakertools.OutputIncisedRestRhythmMaker(
...     prefix_talea=[7, 8],
...     prefix_lengths=[2],
...     suffix_talea=[3],
...     suffix_lengths=[4],
...     talea_denominator=32)
```

Configuration at initialization and then call on arbitrary divisions:

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...     divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.OutputIncisedRhythmMaker`
- `rhythmmakertools.IncisedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`OutputIncisedRestRhythmMaker.storage_format`

Output-incised rest rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.OutputIncisedRestRhythmMaker(
  prefix_talea=[7, 8],
  prefix_lengths=[2],
  suffix_talea=[3],
  suffix_lengths=[4],
  talea_denominator=32,
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

Returns string.

## Methods

`OutputIncisedRestRhythmMaker.new(**kwargs)`

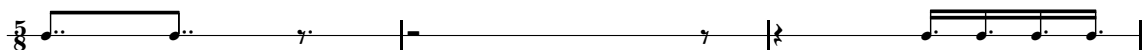
Create new output-incised rest rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(prefix_talea=[7])
```

```
>>> print new_maker.storage_format
rhythmmakertools.OutputIncisedRestRhythmMaker(
  prefix_talea=[7],
  prefix_lengths=[2],
  suffix_talea=[3],
  suffix_lengths=[4],
  talea_denominator=32,
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False
)
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new output-incised rest rhythm-maker.

`OutputIncisedRestRhythmMaker.reverse()`

Reverse output-incised rest rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.OutputIncisedRestRhythmMaker(
  prefix_talea=[8, 7],
```

```

prefix_lengths=[2],
suffix_talea=[3],
suffix_lengths=[4],
talea_denominator=32,
prolation_addenda=[],
secondary_divisions=[],
decrease_durations_monotonically=False,
tie_rests=False,
beam_each_cell=False,
beam_cells_together=False
)

```

```

>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)

```

```
>>> show(staff)
```



Returns new output-incised rest rhythm-maker.

## Special methods

`(IncisedRhythmMaker).__call__(divisions, seeds=None)`

Call incised rhythm-maker on *divisions*.

Returns list of tuplets or return list of leaf lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

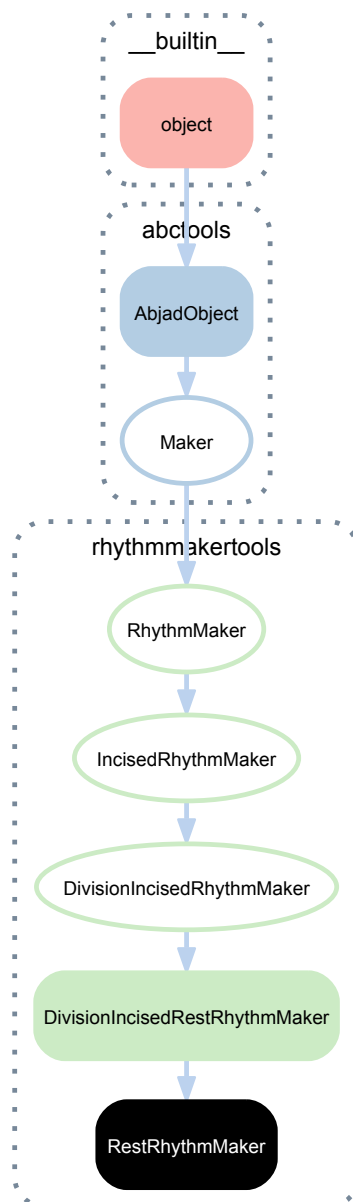
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

23.2.10 `rhythmmakertools.RestRhythmMaker`

**class** `rhythmmakertools.RestRhythmMaker` (*forbidden\_written\_duration=None*)  
 Rest rhythm-maker.

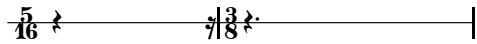
**Example 1:**

```
>>> maker = rhythmmakertools.RestRhythmMaker()
```

Initialize and then call on arbitrary divisions:

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)

>>> show(staff)
```

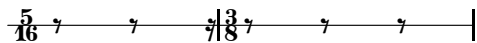


**Example 2.** Forbid written durations greater than or equal to a half note:

```
>>> maker = rhythmmakertools.RestRhythmMaker(
...     forbidden_written_duration=Duration(1, 4))

>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.DivisionIncisedRestRhythmMaker`
- `rhythmmakertools.DivisionIncisedRhythmMaker`
- `rhythmmakertools.IncisedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`RestRhythmMaker.storage_format`

Rest rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.RestRhythmMaker(
    forbidden_written_duration=durationtools.Duration(1, 4)
)
```

Returns string.

## Methods

`RestRhythmMaker.new(**kwargs)`

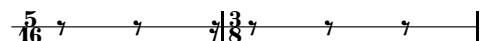
Create new rest rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new()
```

```
>>> print new_maker.storage_format
rhythmmakertools.RestRhythmMaker(
    forbidden_written_duration=durationtools.Duration(1, 4)
)
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new rest rhythm-maker.

`RestRhythmMaker.reverse()`

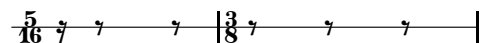
Reverse rest rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.RestRhythmMaker(
    forbidden_written_duration=durationtools.Duration(1, 4)
)
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

```
>>> show(staff)
```



Returns new rest rhythm-maker.

## Special methods

`(IncisedRhythmMaker).__call__(divisions, seeds=None)`

Call incised rhythm-maker on *divisions*.

Returns list of tuplets or return list of leaf lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

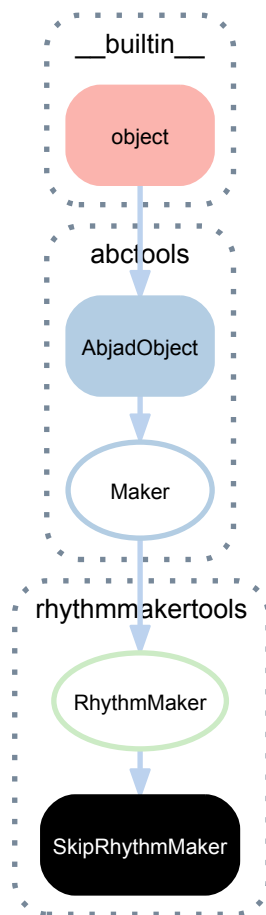
Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

### 23.2.11 rhythmtools.SkipRhythmMaker



**class** `rhythmmakertools.SkipRhythmMaker`  
Skip rhythm-maker:

```
>>> maker = rhythmmakertools.SkipRhythmMaker()
```

Initialize and then call on arbitrary divisions:

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> leaf_lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```

$\frac{1}{4}$  ———  $\frac{3}{16}$  ———  $\frac{5}{8}$  ——— |

Usage follows the two-step instantiate-then-call pattern shown here.

#### Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`

- `__builtin__.object`

## Read-only properties

`SkipRhythmMaker.storage_format`

Skip rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.SkipRhythmMaker()
```

Returns string.

## Methods

`SkipRhythmMaker.new(**kwargs)`

Create new skip rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new()
```

```
>>> print new_maker.storage_format
rhythmmakertools.SkipRhythmMaker()
```

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> leaf_lists = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



A musical staff with a single line. It contains three notes: a quarter note with a '1' above it, a triplet of eighth notes with a '3' above it, and a quarter note with a '5' above it. The notes are connected by a horizontal line.

Returns new skip rhythm-maker.

`SkipRhythmMaker.reverse()`

Reverse skip rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.SkipRhythmMaker()
```

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



A musical staff with a single line. It contains three notes: a quarter note with a '1' above it, a triplet of eighth notes with a '3' above it, and a quarter note with a '5' above it. The notes are connected by a horizontal line.

Returns new skip rhythm-maker.



## Special methods

`SkipRhythmMaker.__call__(divisions, seeds=None)`

Call skip rhythm-maker on *divisions*.

Returns list of skips.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

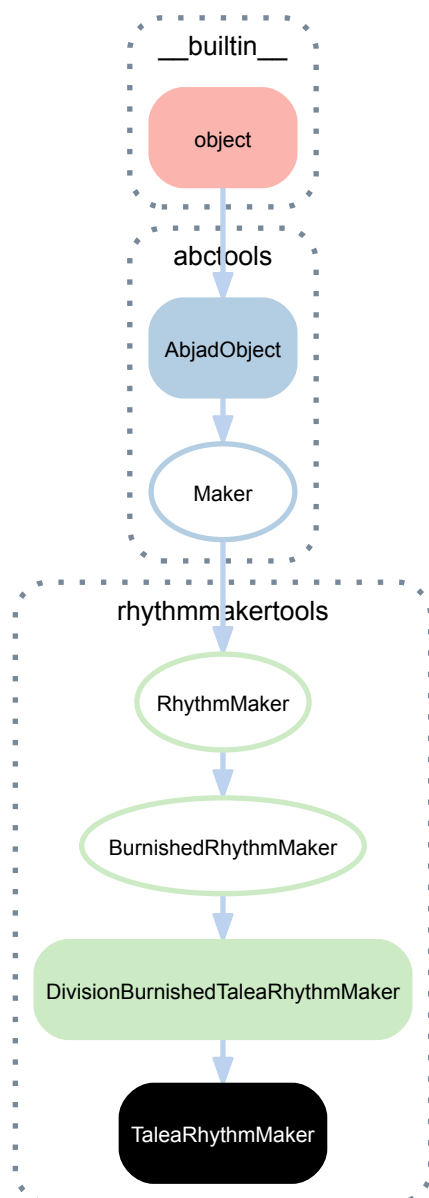
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

23.2.12 `rhythmmakertools.TaleaRhythmMaker`

```

class rhythmmakertools.TaleaRhythmMaker (talea=None,          talea_denominator=None,
                                           prolation_addenda=None,      sec-
                                           ondary_divisions=None,    talea_helper=None,
                                           prolation_addenda_helper=None,
                                           secondary_divisions_helper=None,
                                           beam_each_cell=False,
                                           beam_cells_together=False,
                                           tie_split_notes=False)

```

Talea rhythm-maker.

**Example 1.** Basic usage:

```

>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=[-1, 4, -2, 3],
...     talea_denominator=16,
...     prolation_addenda=[3, 4])

```

```

>>> divisions = [(2, 8), (5, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)

```

```
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



**Example 2.** Tie split notes.

```
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=[5],
...     talea_denominator=16,
...     tie_split_notes=True)
```

```
>>> divisions = [(2, 8), (2, 8), (2, 8), (2, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.DivisionBurnishedTaleaRhythmMaker`
- `rhythmmakertools.BurnishedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`TaleaRhythmMaker.storage_format`

Talea rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.TaleaRhythmMaker(
    talea=[5],
    talea_denominator=16,
    prolation_addenda=[],
    secondary_divisions=[],
    beam_each_cell=False,
    beam_cells_together=False,
    tie_split_notes=True
)
```

Returns string.

## Methods

TaleaRhythmMaker.**new** (\*\*kwargs)

Create new talea rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(prolation_addenda=[1])
```

```
>>> print new_maker.storage_format
rhythmmakertools.TaleaRhythmMaker(
  talea=[5],
  talea_denominator=16,
  prolation_addenda=[1],
  secondary_divisions=[],
  beam_each_cell=False,
  beam_cells_together=False,
  tie_split_notes=True
)
```

```
>>> divisions = [(2, 8), (5, 8)]
>>> music = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Returns new talea rhythm-maker.

TaleaRhythmMaker.**reverse**()

Reverse talea rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.TaleaRhythmMaker(
  talea=[5],
  talea_denominator=16,
  prolation_addenda=[],
  secondary_divisions=[],
  beam_each_cell=False,
  beam_cells_together=False,
  tie_split_notes=True
)
```

```
>>> divisions = [(2, 8), (5, 8)]
>>> music = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...         divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

```
>>> show(staff)
```



Returns new talea rhythm-maker.

## Special methods

`(BurnishedRhythmMaker).__call__(divisions, seeds=None)`

Call burnished rhythm-maker on *divisions*.

Returns either list of tuplets or else list of note-lists.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

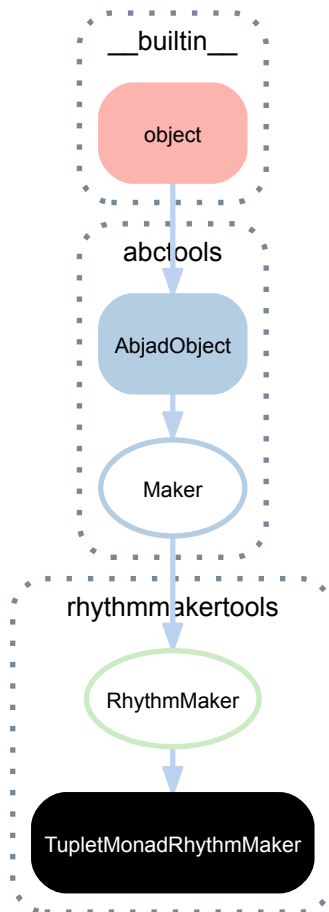
Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

### 23.2.13 `rhythmmakertools.TupletMonadRhythmMaker`



**class** `rhythmmakertools.TupletMonadRhythmMaker` (*beam\_each\_cell=False*,  
   *beam\_cells\_together=False*)

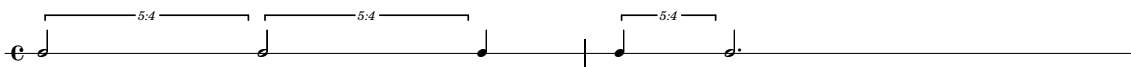
Tuplet monad rhythm-maker:

```
>>> maker = rhythmmakertools.TupletMonadRhythmMaker()
```

Initialize and then call on arbitrary divisions:

```
>>> divisions = [(2, 5), (2, 5), (1, 4), (1, 5), (3, 4)]
>>> tuplet_lists = maker(divisions)
>>> tuplets = sequencetools.flatten_sequence(tuplet_lists)
>>> staff = stafftools.RhythmicStaff(tuplets)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

## Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`TupletMonadRhythmMaker.storage_format`

Tuplet monad rhythm-maker storage format:

```
>>> print maker.storage_format
rhythmmakertools.TupletMonadRhythmMaker(
    beam_each_cell=False,
    beam_cells_together=False
)
```

Returns string.

## Methods

`TupletMonadRhythmMaker.new(**kwargs)`

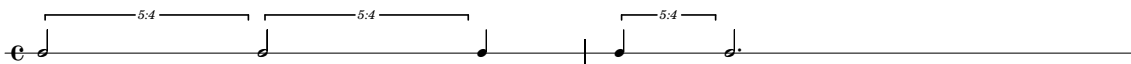
Create new tuplet monad rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new()
```

```
>>> print new_maker.storage_format
rhythmmakertools.TupletMonadRhythmMaker(
    beam_each_cell=False,
    beam_cells_together=False
)
```

```
>>> divisions = [(2, 5), (2, 5), (1, 4), (1, 5), (3, 4)]
>>> tuplet_lists = new_maker(divisions)
>>> tuplets = sequencetools.flatten_sequence(tuplet_lists)
>>> staff = stafftools.RhythmicStaff(tuplets)
```

```
>>> show(staff)
```



Returns new tuplet monad rhythm-maker.

`TupletMonadRhythmMaker.reverse()`

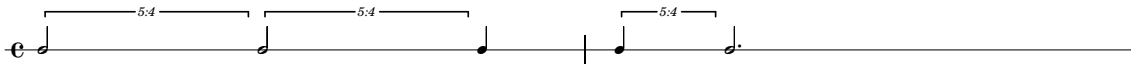
Reverse tuplet monad rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print reversed_maker.storage_format
rhythmmakertools.TupletMonadRhythmMaker (
  beam_each_cell=False,
  beam_cells_together=False
)
```

```
>>> divisions = [(2, 5), (2, 5), (1, 4), (1, 5), (3, 4)]
>>> tuplet_lists = reversed_maker(divisions)
>>> tuplets = sequencetools.flatten_sequence(tuplet_lists)
>>> staff = stafftools.RhythmicStaff(tuplets)
```

```
>>> show(staff)
```



Returns new tuplet monad rhythm-maker.

## Special methods

`TupletMonadRhythmMaker.__call__(divisions, seeds=None)`

Call tuplet monad rhythm-maker on *divisions*.

Returns list of tuplets.

`(RhythmMaker).__eq__(expr)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(RhythmMaker).__repr__()`

Rhythm-maker interpreter representation.

Returns string.

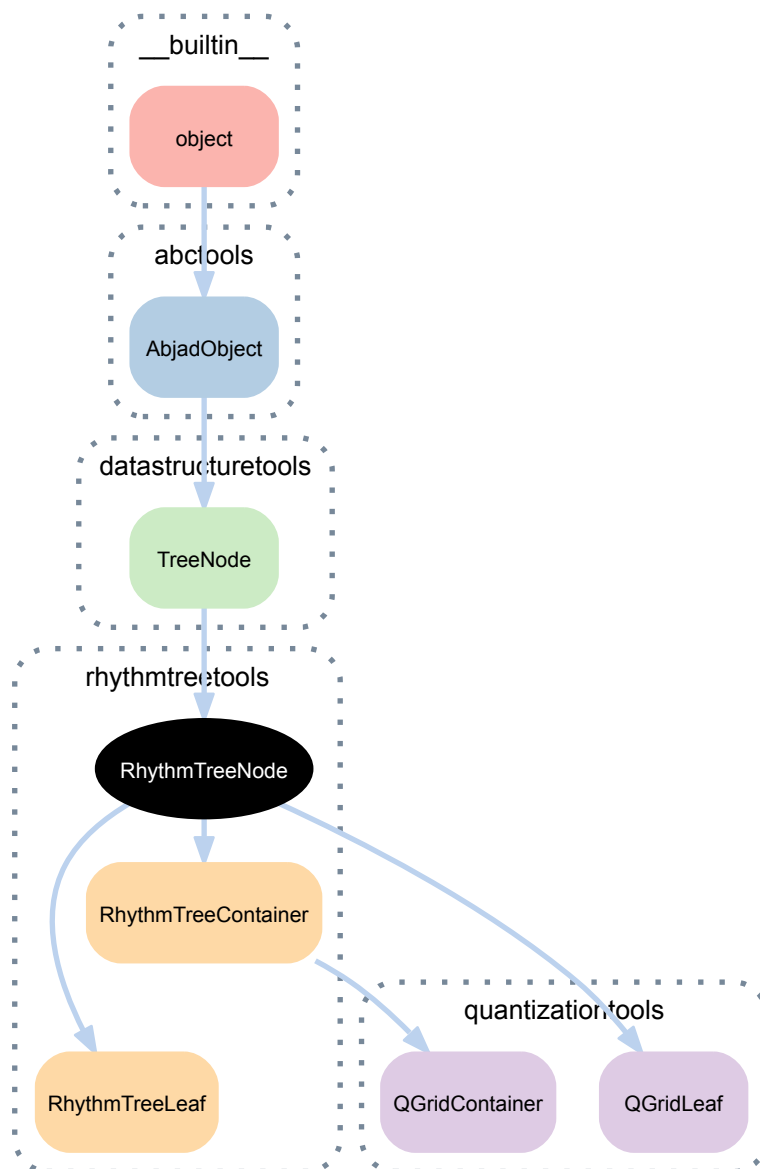




# RHYTHMTREETOOLS

## 24.1 Abstract classes

### 24.1.1 `rhythmtreetools.RhythmTreeNode`



**class** `rhythmtreetools.RhythmTreeNode` (*preprolated\_duration=1, name=None*)  
Abstract base class of nodes in a rhythm tree structure.

## Bases

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(`TreeNode`) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(`TreeNode`) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

`RhythmTreeNode` **.duration**

The prolated preprolated\_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

(*TreeNode*) **.graph\_order**

RhythmTreeNode.**graphviz\_format**

RhythmTreeNode.**graphviz\_graph**

(*TreeNode*) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

RhythmTreeNode.**parentage\_ratios**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated\_duration of the root node, and subsequent items are pairs of the preprolated duration of the next node in the parentage chain and the total preprolated\_duration of that node and its siblings:

```
>>> a = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)

>>> a.extend([b, c])
>>> b.extend([d, e])

>>> a.parentage_ratios
(Duration(1, 1),)

>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))

>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))

>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))

>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Returns tuple.

RhythmTreeNode.**pretty\_rtm\_format**

The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
  (1 (
    1
    1))
  (1 (
    1
    1))))
```

Returns string.

RhythmTreeNode.**prolation**

RhythmTreeNode.**prolations**

(TreeNode).**proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeNode).**.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

**RhythmTreeNode.rtm\_format**

The node's RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> tree.rtm_format
'(1 ((1 (1 1)) (1 (1 1))))'
```

Returns string.

**RhythmTreeNode.start\_offset**

The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Returns Offset instance.

**RhythmTreeNode.stop\_offset**

The stopping offset of a node in a rhythm-tree relative the root.

## Read/write properties

(TreeNode).**name**

**RhythmTreeNode.preprolated\_duration**

The node's preprolated\_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(
...     preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Returns int.

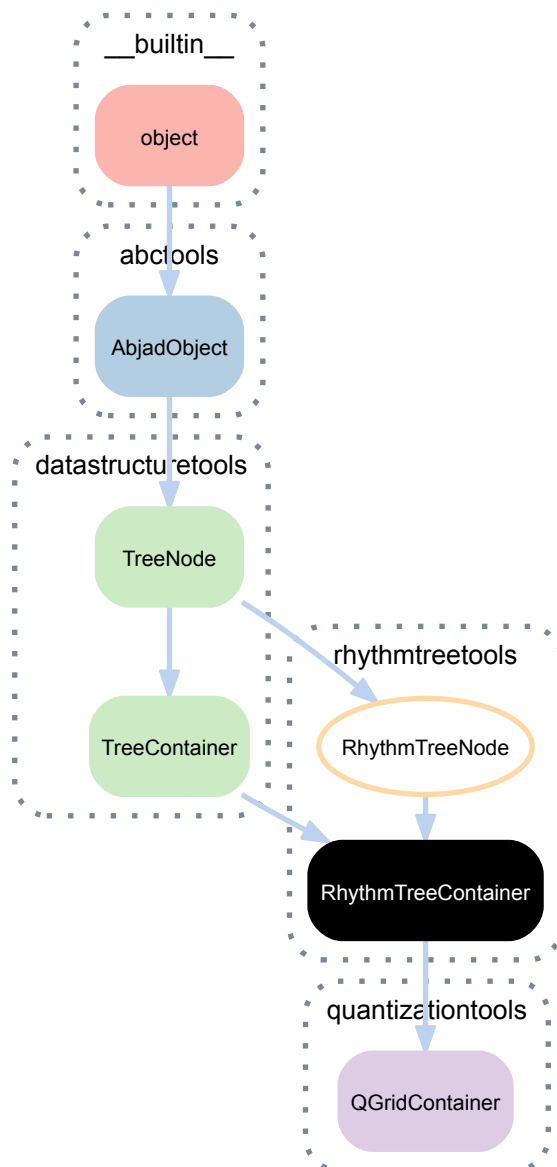
## Special methods

RhythmTreeNode.**\_\_call\_\_**(pulse\_duration)

```
(TreeNode) .__copy__ (*args)
(TreeNode) .__deepcopy__ (*args)
(TreeNode) .__eq__ (expr)
(TreeNode) .__getstate__ ()
(TreeNode) .__ne__ (expr)
(TreeNode) .__repr__ ()
(TreeNode) .__setstate__ (state)
```

## 24.2 Concrete classes

### 24.2.1 `rhythmtreetools.RhythmTreeContainer`



```
class rhythmtreetools.RhythmTreeContainer (children=None,      preprolated_duration=1,
                                           name=None)
```

A container node in a rhythm tree structure:

```
>>> container = rhythmtreetools.RhythmTreeContainer(
...     preprolated_duration=1, children=[])
>>> container
RhythmTreeContainer(
  preprolated_duration=Duration(1, 1)
)
```

Similar to Abjad containers, *RhythmTreeContainer* supports a list interface, and can be appended, extended, indexed and so forth by other *RhythmTreeNode* subclasses:

```
>>> leaf_a = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=1)
>>> leaf_b = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=2)
>>> container.extend([leaf_a, leaf_b])
>>> container
RhythmTreeContainer(
  children=(
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(2, 1),
      is_pitched=True
    )
  ),
  preprolated_duration=Duration(1, 1)
)
```

```
>>> another_container = rhythmtreetools.RhythmTreeContainer(
...     preprolated_duration=2)
>>> another_container.append(
...     rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3))
>>> another_container.append(container[1])
>>> container.append(another_container)
>>> container
RhythmTreeContainer(
  children=(
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeContainer(
      children=(
        RhythmTreeLeaf(
          preprolated_duration=Duration(3, 1),
          is_pitched=True
        ),
        RhythmTreeLeaf(
          preprolated_duration=Duration(2, 1),
          is_pitched=True
        )
      ),
      preprolated_duration=Duration(2, 1)
    )
  ),
  preprolated_duration=Duration(1, 1)
)
```

Call *RhythmTreeContainer* with a *preprolated\_duration* to generate a tuplet structure:

```
>>> container((1, 4))
[FixedDurationTuplet(1/4, [c'8, {@ 5:4 c'8., c'8 @})]
```

Returns *RhythmTreeContainer* instance.

## Bases

- `rhythmtreetools.RhythmTreeNode`
- `datastructuretools.TreeContainer`

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(`TreeContainer`) **.children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

(`TreeNode`) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(`TreeNode`) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```



```

>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g

```

Returns dictionary.

(RhythmTreeNode).**duration**

The prolated preprolated\_duration of the node:

```

>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]

```

```

>>> tree.duration
Duration(1, 1)

```

```

>>> tree[1].duration
Duration(1, 2)

```

```

>>> tree[1][1].duration
Duration(1, 4)

```

Return *Duration* instance.

(TreeNode).**graph\_order**

(RhythmTreeNode).**graphviz\_format**

RhythmTreeContainer.**graphviz\_graph**

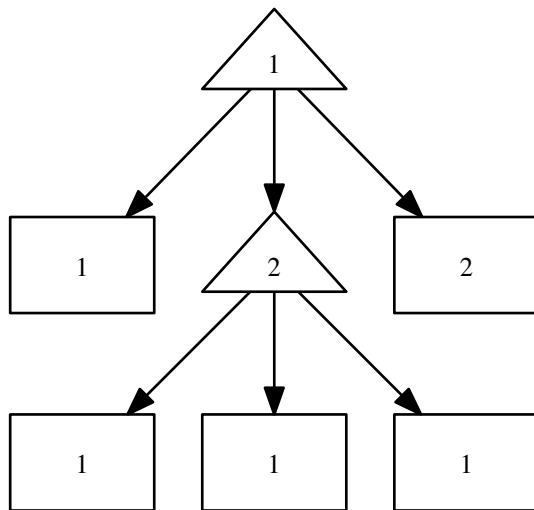
The GraphvizGraph representation of the RhythmTreeContainer:

```

>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> graph = tree.graphviz_graph
>>> print graph.graphviz_format
digraph G {
    node_0 [label=1,
            shape=triangle];
    node_1 [label=1,
            shape=box];
    node_2 [label=2,
            shape=triangle];
    node_3 [label=1,
            shape=box];
    node_4 [label=1,
            shape=box];
    node_5 [label=1,
            shape=box];
    node_6 [label=2,
            shape=box];
    node_0 -> node_1;
    node_0 -> node_2;
    node_0 -> node_6;
    node_2 -> node_3;
    node_2 -> node_4;
    node_2 -> node_5;
}

```

```
>>> iotools.graph(graph)
```



Return *GraphvizGraph* instance.

(*TreeNode*) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeContainer*) **.leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

(*TreeContainer*) **.nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(*TreeNode*).**.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(*RhythmTreeNode*).**.parentage\_ratios**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated\_duration of the root node, and subsequent items are pairs of the preprolated duration of the next node in the parentage chain and the total preprolated\_duration of that node and its siblings:

```
>>> a = rhythmtreertools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreertools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreertools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreertools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreertools.RhythmTreeLeaf(preprolated_duration=5)
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.parentage_ratios
(Duration(1, 1),)
```

```
>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))
```

```
>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))
```

```
>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))
```

```
>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Returns tuple.

(RhythmTreeNode).**.pretty\_rtm\_format**

The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
  (1 (
    1
    1))
  (1 (
    1
    1))))
```

Returns string.

(RhythmTreeNode).**.prolation**

(RhythmTreeNode).**.prolations**

(TreeNode).**.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeNode).**.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

`RhythmTreeContainer.rtm_format`

The node's RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> tree.rtm_format
'(1 ((1 (1 1)) (1 (1 1))))'
```

Returns string.

`(RhythmTreeNode).start_offset`

The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Returns *Offset* instance.

`(RhythmTreeNode).stop_offset`

The stopping offset of a node in a rhythm-tree relative the root.

## Read/write properties

`(TreeNode).name`

`(RhythmTreeNode).preprolated_duration`

The node's `preprolated_duration` in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(
...     preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Returns int.

## Methods

`(TreeContainer).append(node)`

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.extend** (*expr*)  
 Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.index** (*node*)  
 Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) **.insert** (*i*, *node*)  
 Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.pop** (*i*=-1)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) **.remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

`RhythmTreeContainer.__add__(expr)`

Concatenate containers self and *expr*. The operation `c = a + b` returns a new `RhythmTreeContainer` *c* with the content of both *a* and *b*, and a `preprolated_duration` equal to the sum of the durations of *a* and *b*. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand:

```
>>> a = rhythmtreetools.RhythmTreeParser() ('(1 (1 1 1))') [0]
>>> b = rhythmtreetools.RhythmTreeParser() ('(2 (3 4))') [0]
```

```
>>> c = a + b
```

```
>>> c.preprolated_duration
Duration(3, 1)
```

```
>>> c
RhythmTreeContainer(
  children=(
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(3, 1),
      is_pitched=True
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(4, 1),
      is_pitched=True
    )
  ),
  preprolated_duration=Duration(3, 1)
)
```

Returns new `RhythmTreeContainer`.

`RhythmTreeContainer.__call__(pulse_duration)`

Generate Abjad score components:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser() (rtm) [0]
```

```
>>> tree((1, 4))
[FixedDurationTuplet(1/4, [c'16, {@ 3:2 c'16, c'16, c'16 @}, c'8])]
```

Returns sequence of components.



(TreeContainer).**\_\_contains\_\_**(*expr*)  
 True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode).**\_\_copy\_\_**(\*args)

(TreeNode).**\_\_deepcopy\_\_**(\*args)

(TreeContainer).**\_\_delitem\_\_**(*i*)  
 Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

RhythmTreeContainer.**\_\_eq\_\_**(*expr*)  
 True if type, preprolotted\_duration and children are equivalent. Otherwise False.

Returns boolean.

(TreeContainer).**\_\_getitem\_\_**(*i*)  
 Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(*TreeNode*).**\_\_getstate\_\_**()

(*TreeContainer*).**\_\_iter\_\_**()

(*TreeContainer*).**\_\_len\_\_**()

Returns nonnegative integer number of nodes in container.

(*TreeNode*).**\_\_ne\_\_**(*expr*)

(*TreeNode*).**\_\_repr\_\_**()

RhythmTreeContainer.**\_\_setitem\_\_**(*i*, *expr*)

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = rhythmtreetools.RhythmTreeContainer()
>>> b = rhythmtreetools.RhythmTreeLeaf()
>>> c = rhythmtreetools.RhythmTreeLeaf()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

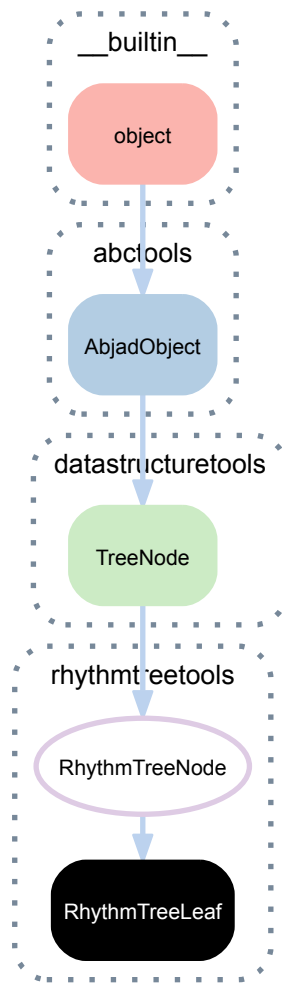
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

(*TreeNode*).**\_\_setstate\_\_**(*state*)

## 24.2.2 `rhythmtreetools.RhythmTreeLeaf`



**class** `rhythmtreetools.RhythmTreeLeaf` (*preprolated\_duration=1*, *is\_pitched=True*, *name=None*)

A leaf node in a rhythm tree:

```
>>> leaf = rhythmtreetools.RhythmTreeLeaf(
...     preprolated_duration=5, is_pitched=True)
>>> leaf
RhythmTreeLeaf(
    preprolated_duration=Duration(5, 1),
    is_pitched=True
)
```

Call with a pulse `preprolated_duration` to generate Abjad leaf objects:

```
>>> result = leaf((1, 8))
>>> result
Selection(Note("c'2"), Note("c'8"))
```

Generates rests when called, if *is\_pitched* is `False`:

```
>>> rhythmtreetools.RhythmTreeLeaf(
...     preprolated_duration=7, is_pitched=False)((1, 16))
Selection(Rest('r4..'))
```

Return *RhythmTreeLeaf*.

### Bases

- `rhythmtreetools.RhythmTreeNode`

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `(TreeNode).depth`

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

### `(TreeNode).depthwise_inventory`

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

### `(RhythmTreeNode).duration`

The prolated preprolated\_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

(*TreeNode*) **.graph\_order**

(*RhythmTreeNode*) **.graphviz\_format**

*RhythmTreeLeaf*.**graphviz\_graph**

(*TreeNode*) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(*RhythmTreeNode*) **.parentage\_ratios**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated\_duration of the root node, and subsequent items are pairs of the preprolated duration of the next node in the parentage chain and the total preprolated\_duration of that node and its siblings:

```
>>> a = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)

>>> a.extend([b, c])
>>> b.extend([d, e])

>>> a.parentage_ratios
(Duration(1, 1),)

>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))

>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))

>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))

>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Returns tuple.

(RhythmTreeNode).**.pretty\_rtm\_format**

The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
  (1 (
    1
    1))
  (1 (
    1
    1))))
```

Returns string.

(RhythmTreeNode).**.prolation**

(RhythmTreeNode).**.prolations**

(TreeNode).**.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeNode).**.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

*RhythmTreeLeaf*.**rtm\_format**

The node's RTM format:

```
>>> rhythmtreetools.RhythmTreeLeaf(1, is_pitched=True).rtm_format
'1'
>>> rhythmtreetools.RhythmTreeLeaf(5, is_pitched=False).rtm_format
'-5'
```

Returns string.

(*RhythmTreeNode*).**.start\_offset**

The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Returns *Offset* instance.

(*RhythmTreeNode*).**.stop\_offset**

The stopping offset of a node in a rhythm-tree relative the root.

## Read/write properties

*RhythmTreeLeaf*.**is\_pitched**

True if leaf is pitched:

```
>>> leaf = rhythmtreetools.RhythmTreeLeaf()
>>> leaf.is_pitched
True
```

```
>>> leaf.is_pitched = False
>>> leaf.is_pitched
False
```

Returns boolean.

(*TreeNode*).**.name**

(*RhythmTreeNode*).**.preprolated\_duration**

The node's preprolated\_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(
...     preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Returns int.

## Special methods

`RhythmTreeLeaf.__call__(pulse_duration)`

Generate Abjad score components:

```
>>> leaf = rhythmtreetools.RhythmTreeLeaf(5)
>>> leaf((1, 4))
Selection(Note("c'1"), Note("c'4"))
```

Returns sequence of components.

`(TreeNode) .__copy__(*args)`

`(TreeNode) .__deepcopy__(*args)`

`RhythmTreeLeaf.__eq__(expr)`

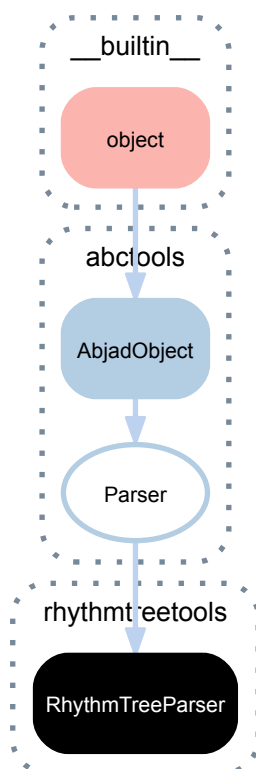
`(TreeNode) .__getstate__()`

`(TreeNode) .__ne__(expr)`

`(TreeNode) .__repr__()`

`(TreeNode) .__setstate__(state)`

### 24.2.3 `rhythmtreetools.RhythmTreeParser`





**class** `rhythmtreetools.RhythmTreeParser` (*debug=False*)  
 Parses RTM-style rhythm syntax:

```
>>> parser = rhythmtreetools.RhythmTreeParser()
```

```
>>> rtm = '(1 (1 (2 (1 -1 1)) -2))'
>>> result = parser(rtm)[0]
>>> result
RhythmTreeContainer(
  children=(
    RhythmTreeLeaf(
      preprolated_duration=Duration(1, 1),
      is_pitched=True
    ),
    RhythmTreeContainer(
      children=(
        RhythmTreeLeaf(
          preprolated_duration=Duration(1, 1),
          is_pitched=True
        ),
        RhythmTreeLeaf(
          preprolated_duration=Duration(1, 1),
          is_pitched=False
        ),
        RhythmTreeLeaf(
          preprolated_duration=Duration(1, 1),
          is_pitched=True
        )
      ),
      preprolated_duration=Duration(2, 1)
    ),
    RhythmTreeLeaf(
      preprolated_duration=Duration(2, 1),
      is_pitched=False
    )
  ),
  preprolated_duration=Duration(1, 1)
)
```

```
>>> result.rtm_format
'(1 (1 (2 (1 -1 1)) -2))'
```

Returns *RhythmTreeParser* instance.

## Bases

- `abctools.Parser`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Parser).debug`  
 True if the parser runs in debugging mode.

`(Parser).lexer`  
 The parser's PLY Lexer instance.

`RhythmTreeParser.lexer_rules_object`

`(Parser).logger`  
 The parser's Logger instance.

`(Parser).logger_path`  
 The output path for the parser's logfile.

`(Parser).output_path`  
The output path for files associated with the parser.

`(Parser).parser`  
The parser's PLY LRPParser instance.

`RhythmTreeParser.parser_rules_object`

`(Parser).pickle_path`  
The output path for the parser's pickled parsing tables.

## Methods

`RhythmTreeParser.p_container__LPAREN__DURATION__node_list_closed__RPAREN(p)`  
container : LPAREN DURATION node\_list\_closed RPAREN

`RhythmTreeParser.p_error(p)`

`RhythmTreeParser.p_leaf__INTEGER(p)`  
leaf : DURATION

`RhythmTreeParser.p_node__container(p)`  
node : container

`RhythmTreeParser.p_node__leaf(p)`  
node : leaf

`RhythmTreeParser.p_node_list__node_list__node_list_item(p)`  
node\_list : node\_list node\_list\_item

`RhythmTreeParser.p_node_list__node_list_item(p)`  
node\_list : node\_list\_item

`RhythmTreeParser.p_node_list_closed__LPAREN__node_list__RPAREN(p)`  
node\_list\_closed : LPAREN node\_list RPAREN

`RhythmTreeParser.p_node_list_item__node(p)`  
node\_list\_item : node

`RhythmTreeParser.p_toplevel__EMPTY(p)`  
toplevel :

`RhythmTreeParser.p_toplevel__toplevel__node(p)`  
toplevel : toplevel node

`RhythmTreeParser.t_DURATION(t)`  
-?[1-9]d\*/([1-9]d\*)?

`RhythmTreeParser.t_error(t)`

`RhythmTreeParser.t_newline(t)`  
n+

`(Parser).tokenize(input_string)`  
Tokenize *input string* and print results.

## Special methods

`(Parser).__call__(input_string)`  
Parse *input\_string* and return result.

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

## 24.3 Functions

### 24.3.1 `rhythmtreetools.parse_rtm_syntax`

`rhythmtreetools.parse_rtm_syntax(rtm)`  
 Parse RTM syntax:

```
>>> rtm = '(1 (1 (1 (1 1)) 1))'
>>> rhythmtreetools.parse_rtm_syntax(rtm)
FixedDurationTuplet(1/4, [c'8, c'16, c'16, c'8])
```

Also supports fractional durations:

```
>>> rtm = '(3/4 (1 1/2 (4/3 (1 -1/2 1))))'
>>> rhythmtreetools.parse_rtm_syntax(rtm)
FixedDurationTuplet(3/16, [c'8, c'16, {@ 15:8 c'8, r16, c'8 @}])
>>> f(_)
\tweak #'text #tuplet-number::calc-fraction-text
\times 9/17 {
  c'8
  c'16
  \times 8/15 {
    c'8
    r16
    c'8
  }
}
```

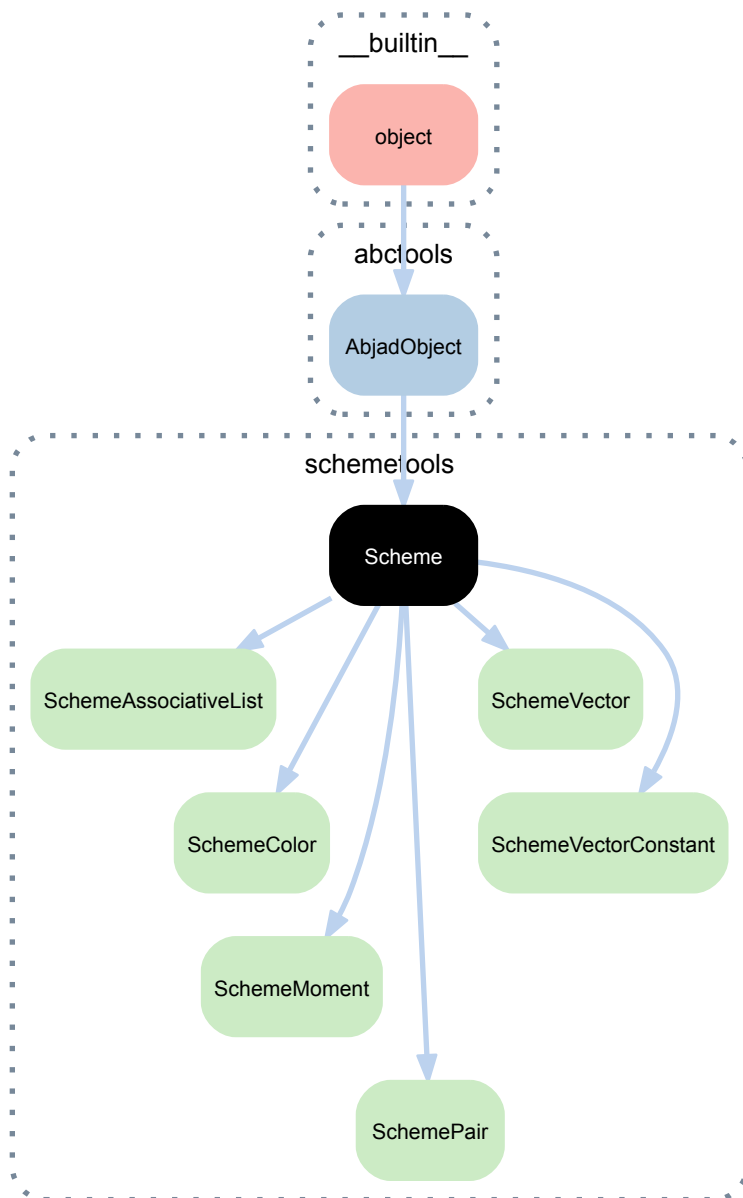
Return *FixedDurationTuplet* or *Container* instance.



## SCHEMETOOLS

### 25.1 Concrete classes

#### 25.1.1 schemetools.Scheme



**class** `schemetools.Scheme` (*\*args, \*\*kwargs*)  
 Abjad model of Scheme code.

```
>>> scheme = schemetools.Scheme(True)
>>> scheme.lilypond_format
'##t'
```

Scheme can represent nested structures:

```
>>> scheme = schemetools.Scheme(
...     ('left', (1, 2, False)), ('right', (1, 2, 3.3)))
>>> scheme.lilypond_format
'##((left (1 2 #f)) (right (1 2 3.3)))'
```

Scheme wraps variable-length arguments into a tuple:

```
>>> scheme_1 = schemetools.Scheme(1, 2, 3)
>>> scheme_2 = schemetools.Scheme((1, 2, 3))
>>> scheme_1.lilypond_format == scheme_2.lilypond_format
True
```

Scheme also takes an optional *quoting* keyword, by which Scheme's various quote, unquote, unquote-splicing characters can be prepended to the formatted result:

```
>>> scheme = schemetools.Scheme((1, 2, 3), quoting="'##")
>>> scheme.lilypond_format
"##'##(1 2 3)"
```

Scheme can also force quotes around strings which contain no whitespace:

```
>>> scheme = schemetools.Scheme('nospaces', force_quotes=True)
>>> f(scheme)
#"nospaces"
```

The above is useful in certain override situations, as LilyPond's Scheme interpreter will treat unquoted strings as symbols rather than strings.

Scheme is immutable.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`Scheme.force_quotes`

`Scheme.lilypond_format`

Hash-mark-prepended format of Scheme:

```
>>> scheme = schemetools.Scheme(True)
>>> scheme.lilypond_format
'##t'
```

Returns string.

`Scheme.storage_format`

Scheme storage format.

Returns string.

## Class methods

`Scheme.format_scheme_value (value, force_quotes=False)`  
 Format *value* as Scheme would:

```
>>> schemetools.Scheme.format_scheme_value(1)
'1'
```

```
>>> schemetools.Scheme.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.Scheme.format_scheme_value('bar baz')
'"bar baz"'
```

```
>>> schemetools.Scheme.format_scheme_value([1.5, True, False])
'(1.5 #t #f)'
```

Strings without whitespace can be forcibly quoted via the *force\_quotes* keyword:

```
>>> schemetools.Scheme.format_scheme_value(
...     'foo', force_quotes=True)
'"foo"'
```

Returns string.

## Special methods

`Scheme.__eq__ (expr)`

`(AbjadObject).__ne__ (expr)`

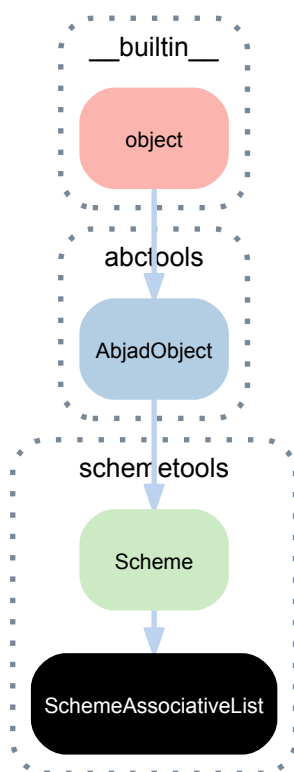
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`Scheme.__repr__ ()`

`Scheme.__str__ ()`

## 25.1.2 schemetools.SchemeAssociativeList



**class** `schemetools.SchemeAssociativeList` (\*args, \*\*kwargs)  
 Abjad model of Scheme associative list:

```
>>> schemetools.SchemeAssociativeList(
...     ('space', 2), ('padding', 0.5))
SchemeAssociativeList((SchemePair(('space', 2)),
    SchemePair(('padding', 0.5))))
```

Scheme associative lists are immutable.

### Bases

- `schemetools.Scheme`
- `abctools.AbjadObject`
- `___builtin___object`

### Read-only properties

`(Scheme).force_quotes`

`(Scheme).lilypond_format`

Hash-mark-prepended format of Scheme:

```
>>> scheme = schemetools.Scheme(True)
>>> scheme.lilypond_format
'##t'
```

Returns string.

`(Scheme).storage_format`

Scheme storage format.

Returns string.



## Class methods

(Scheme).**format\_scheme\_value**(value, force\_quotes=False)  
Format *value* as Scheme would:

```
>>> schemetools.Scheme.format_scheme_value(1)
'1'
```

```
>>> schemetools.Scheme.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.Scheme.format_scheme_value('bar baz')
'"bar baz"'
```

```
>>> schemetools.Scheme.format_scheme_value([1.5, True, False])
'(1.5 #t #f)'
```

Strings without whitespace can be forcibly quoted via the *force\_quotes* keyword:

```
>>> schemetools.Scheme.format_scheme_value(
...     'foo', force_quotes=True)
'"foo"'
```

Returns string.

## Special methods

(Scheme).**\_\_eq\_\_**(expr)

(AbjadObject).**\_\_ne\_\_**(expr)

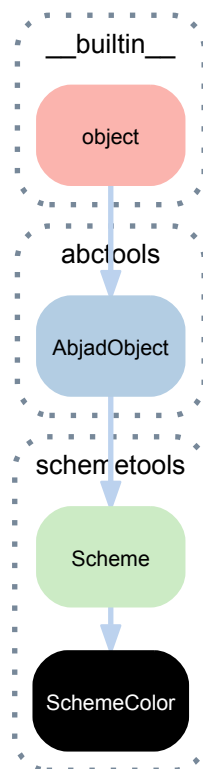
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Scheme).**\_\_repr\_\_**()

(Scheme).**\_\_str\_\_**()

### 25.1.3 schemetools.SchemeColor



**class** `schemetools.SchemeColor(*args, **kwargs)`  
 Abjad model of Scheme color:

```
>>> schemetools.SchemeColor('ForestGreen')
SchemeColor('ForestGreen')
```

Scheme colors are immutable.

#### Bases

- `schemetools.Scheme`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(Scheme).force_quotes`

`(Scheme).lilypond_format`

Hash-mark-prepended format of Scheme:

```
>>> scheme = schemetools.Scheme(True)
>>> scheme.lilypond_format
'##t'
```

Returns string.

`(Scheme).storage_format`

Scheme storage format.

Returns string.

## Class methods

(Scheme) .**format\_scheme\_value** (value, force\_quotes=False)  
Format *value* as Scheme would:

```
>>> schemetools.Scheme.format_scheme_value(1)
'1'
```

```
>>> schemetools.Scheme.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.Scheme.format_scheme_value('bar baz')
'"bar baz"'
```

```
>>> schemetools.Scheme.format_scheme_value([1.5, True, False])
'(1.5 #t #f)'
```

Strings without whitespace can be forcibly quoted via the *force\_quotes* keyword:

```
>>> schemetools.Scheme.format_scheme_value(
...     'foo', force_quotes=True)
'"foo"'
```

Returns string.

## Special methods

(Scheme) .**\_\_eq\_\_** (expr)

(AbjadObject) .**\_\_ne\_\_** (expr)

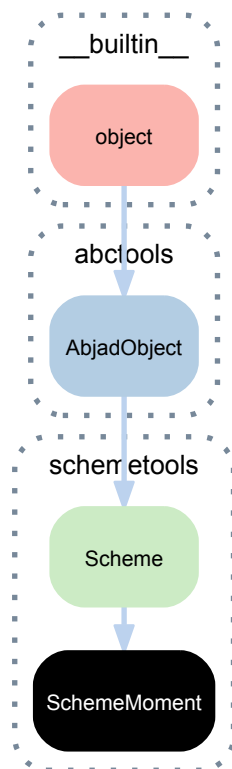
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Scheme) .**\_\_repr\_\_** ()

(Scheme) .**\_\_str\_\_** ()

## 25.1.4 schemetools.SchemeMoment



**class** `schemetools.SchemeMoment` (\*args, \*\*kwargs)  
 Abjad model of LilyPond moment:

```
>>> schemetools.SchemeMoment(1, 68)
SchemeMoment((1, 68))
```

Initialize scheme moments with a single fraction, two integers or another scheme moment.

Scheme moments are immutable.

### Bases

- `schemetools.Scheme`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`SchemeMoment.duration`  
 Duration of scheme moment:

```
>>> scheme_moment = schemetools.SchemeMoment(1, 68)
>>> scheme_moment.duration
Duration(1, 68)
```

Returns duration.

`(Scheme).force_quotes`

`(Scheme).lilypond_format`

Hash-mark-prepended format of Scheme:

```
>>> scheme = schemetools.Scheme(True)
>>> scheme.lilypond_format
'##t'
```

Returns string.

(Scheme).**storage\_format**

Scheme storage format.

Returns string.

## Class methods

(Scheme).**format\_scheme\_value**(value, force\_quotes=False)

Format *value* as Scheme would:

```
>>> schemetools.Scheme.format_scheme_value(1)
'1'
```

```
>>> schemetools.Scheme.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.Scheme.format_scheme_value('bar baz')
'"bar baz"'
```

```
>>> schemetools.Scheme.format_scheme_value([1.5, True, False])
'(1.5 #t #f)'
```

Strings without whitespace can be forcibly quoted via the *force\_quotes* keyword:

```
>>> schemetools.Scheme.format_scheme_value(
...     'foo', force_quotes=True)
'"foo"'
```

Returns string.

## Special methods

SchemeMoment.**\_\_eq\_\_**(arg)

SchemeMoment.**\_\_ge\_\_**(arg)

SchemeMoment.**\_\_gt\_\_**(arg)

SchemeMoment.**\_\_le\_\_**(arg)

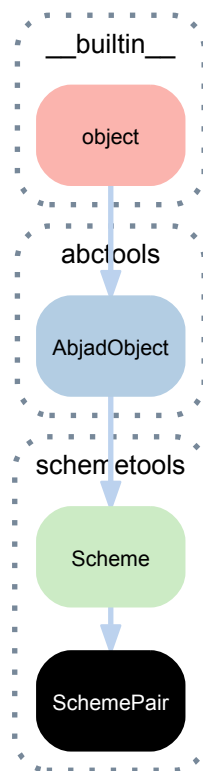
SchemeMoment.**\_\_lt\_\_**(arg)

SchemeMoment.**\_\_ne\_\_**(arg)

SchemeMoment.**\_\_repr\_\_**()

(Scheme).**\_\_str\_\_**()

### 25.1.5 schemetools.SchemePair



**class** `schemetools.SchemePair(*args, **kwargs)`  
A Scheme pair.

```
>>> schemetools.SchemePair('spacing', 4)
SchemePair(('spacing', 4))
```

Initialize Scheme pairs with a tuple, two separate values or another Scheme pair.

Scheme pairs are immutable.

#### Bases

- `schemetools.Scheme`
- `abctools.AbjadObject`
- `___builtin___object`

#### Read-only properties

`(Scheme).force_quotes`

`SchemePair.lilypond_format`

`(Scheme).storage_format`

Scheme storage format.

Returns string.

#### Class methods

`(Scheme).format_scheme_value(value, force_quotes=False)`

Format *value* as Scheme would:

```
>>> schemetools.Scheme.format_scheme_value(1)
'1'
```

```
>>> schemetools.Scheme.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.Scheme.format_scheme_value('bar baz')
'"bar baz"'
```

```
>>> schemetools.Scheme.format_scheme_value([1.5, True, False])
'(1.5 #t #f)'
```

Strings without whitespace can be forcibly quoted via the *force\_quotes* keyword:

```
>>> schemetools.Scheme.format_scheme_value(
...     'foo', force_quotes=True)
'"foo"'
```

Returns string.

## Special methods

(Scheme) .**\_\_eq\_\_**(*expr*)

(AbjadObject) .**\_\_ne\_\_**(*expr*)

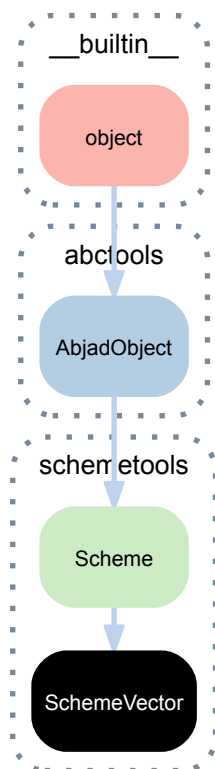
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Scheme) .**\_\_repr\_\_**()

(Scheme) .**\_\_str\_\_**()

### 25.1.6 schemetools.SchemeVector



**class** `schemetools.SchemeVector` (*\*args*)  
 Abjad model of Scheme vector:

```
>>> schemetools.SchemeVector(True, True, False)
SchemeVector((True, True, False))
```

Scheme vectors and Scheme vector constants differ in only their LilyPond input format.

Scheme vectors are immutable.

## Bases

- `schemetools.Scheme`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Scheme).**force\_quotes**

(Scheme).**lilypond\_format**

Hash-mark-prepended format of Scheme:

```
>>> scheme = schemetools.Scheme(True)
>>> scheme.lilypond_format
'##t'
```

Returns string.

(Scheme).**storage\_format**

Scheme storage format.

Returns string.

## Class methods

(Scheme).**format\_scheme\_value**(value, force\_quotes=False)

Format *value* as Scheme would:

```
>>> schemetools.Scheme.format_scheme_value(1)
'1'
```

```
>>> schemetools.Scheme.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.Scheme.format_scheme_value('bar baz')
'"bar baz"'
```

```
>>> schemetools.Scheme.format_scheme_value([1.5, True, False])
'(1.5 #t #f)'
```

Strings without whitespace can be forcibly quoted via the *force\_quotes* keyword:

```
>>> schemetools.Scheme.format_scheme_value(
...     'foo', force_quotes=True)
'"foo"'
```

Returns string.

## Special methods

(Scheme).**\_\_eq\_\_**(expr)

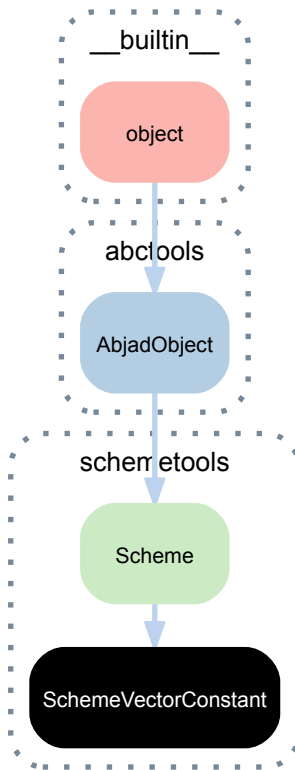


```
(AbjadObject).__ne__(expr)
    True when ID of expr does not equal ID of Abjad object.

    Returns boolean.

(Scheme).__repr__()
(Scheme).__str__()
```

### 25.1.7 schemetools.SchemeVectorConstant



```
class schemetools.SchemeVectorConstant(*args)
    Abjad model of Scheme vector constant:
```

```
>>> schemetools.SchemeVectorConstant(True, True, False)
SchemeVectorConstant((True, True, False))
```

Scheme vectors and Scheme vector constants differ in only their LilyPond input format.

Scheme vector constants are immutable.

#### Bases

- `schemetools.Scheme`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

```
(Scheme).force_quotes
(Scheme).lilypond_format
    Hash-mark-prepended format of Scheme:
```

```
>>> scheme = schemetools.Scheme(True)
>>> scheme.lilypond_format
'##t'
```

Returns string.

(Scheme).**storage\_format**

Scheme storage format.

Returns string.

## Class methods

(Scheme).**format\_scheme\_value**(value, force\_quotes=False)

Format *value* as Scheme would:

```
>>> schemetools.Scheme.format_scheme_value(1)
'1'
```

```
>>> schemetools.Scheme.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.Scheme.format_scheme_value('bar baz')
'"bar baz"'
```

```
>>> schemetools.Scheme.format_scheme_value([1.5, True, False])
'(1.5 #t #f)'
```

Strings without whitespace can be forcibly quoted via the *force\_quotes* keyword:

```
>>> schemetools.Scheme.format_scheme_value(
...     'foo', force_quotes=True)
'"foo"'
```

Returns string.

## Special methods

(Scheme).**\_\_eq\_\_**(expr)

(AbjadObject).**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

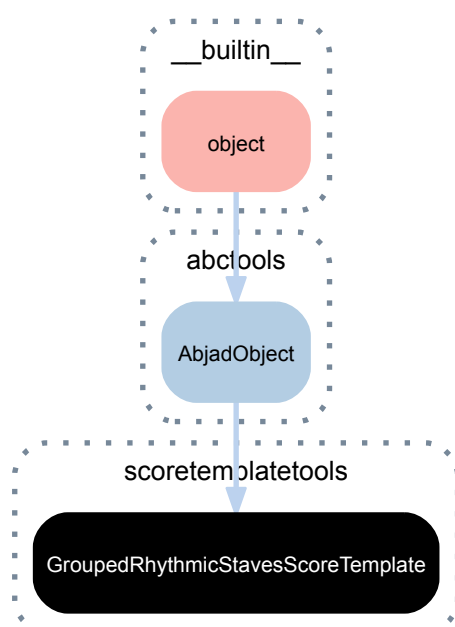
(Scheme).**\_\_repr\_\_**()

(Scheme).**\_\_str\_\_**()

## SCORETEMPLATETOOLS

### 26.1 Concrete classes

#### 26.1.1 scoretemplatetools.GroupedRhythmicStavesScoreTemplate



**class** `scoretemplatetools.GroupedRhythmicStavesScoreTemplate` (*staff\_count=2*)  
Grouped rhythmic staves score template.

```
>>> from abjad.tools.scoretemplatetools import *
>>> template_class = GroupedRhythmicStavesScoreTemplate
```

**Example 1.** One voice per staff:

```
>>> template_1 = template_class(staff_count=4)
```

**Example 2.** More than one voice per staff:

```
>>> template_2 = template_class(staff_count=[2, 1, 2])
```

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`GroupedRhythmicStavesScoreTemplate.staff_count`  
Score template staff count.

```
>>> template_1.staff_count
4
```

Returns nonnegative integer.

## Special methods

`GroupedRhythmicStavesScoreTemplate.__call__()`  
Calls score template.

**Example 1.** Call first template:

```
>>> score_1 = template_1()
```

```
>>> f(score_1)
\context Score = "Grouped Rhythmic Staves Score" <<
  \context StaffGroup = "Grouped Rhythmic Staves Staff Group" <<
    \context RhythmicStaff = "Staff 1" {
      \context Voice = "Voice 1" {
      }
    }
    \context RhythmicStaff = "Staff 2" {
      \context Voice = "Voice 2" {
      }
    }
    \context RhythmicStaff = "Staff 3" {
      \context Voice = "Voice 3" {
      }
    }
    \context RhythmicStaff = "Staff 4" {
      \context Voice = "Voice 4" {
      }
    }
  }
>>
>>
```

**Example 2.** Call second template:

```
>>> score_2 = template_2()
```

```
>>> f(score_2)
\context Score = "Grouped Rhythmic Staves Score" <<
  \context StaffGroup = "Grouped Rhythmic Staves Staff Group" <<
    \context RhythmicStaff = "Staff 1" <<
      \context Voice = "Voice 1-1" {
      }
      \context Voice = "Voice 1-2" {
      }
    >>
    \context RhythmicStaff = "Staff 2" {
      \context Voice = "Voice 2" {
      }
    }
    \context RhythmicStaff = "Staff 3" <<
      \context Voice = "Voice 3-1" {
      }
      \context Voice = "Voice 3-2" {
      }
    >>
  >>
>>
```

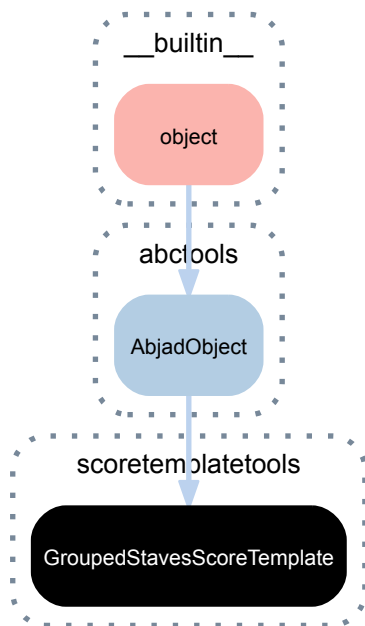
Returns score.

(AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 26.1.2 scoretemplatetools.GroupedStavesScoreTemplate



**class** scoretemplatetools.**GroupedStavesScoreTemplate**(*staff\_count=2*)  
 Grouped staves score template.

```
>>> template_class = scoretemplatetools.GroupedStavesScoreTemplate
>>> template = template_class(staff_count=4)
```

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`GroupedStavesScoreTemplate.__call__()`  
 Calls score template.

```
>>> score = template()
```

```
>>> f(score)
\context Score = "Grouped Staves Score" <<
  \context StaffGroup = "Grouped Staves Staff Group" <<
    \context Staff = "Staff 1" {
```

```

        \context Voice = "Voice 1" {
        }
    }
    \context Staff = "Staff 2" {
        \context Voice = "Voice 2" {
        }
    }
    \context Staff = "Staff 3" {
        \context Voice = "Voice 3" {
        }
    }
    \context Staff = "Staff 4" {
        \context Voice = "Voice 4" {
        }
    }
}
>>
>>

```

Returns score.

(AbjadObject).**\_\_eq\_\_**(*expr*)  
True when ID of *expr* equals ID of Abjad object.

Returns boolean.

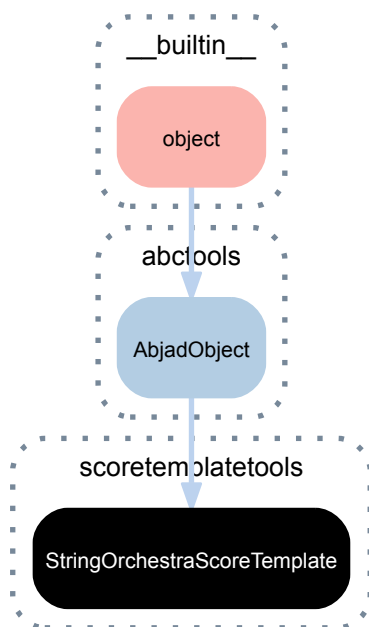
(AbjadObject).**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
Interpreter representation of Abjad object.

Returns string.

### 26.1.3 scoretemplatetools.StringOrchestraScoreTemplate



```

class scoretemplatetools.StringOrchestraScoreTemplate (violin_count=6,          vi-
                                                         ola_count=4, cello_count=3,
                                                         contrabass_count=2)

```

String orchestra score template.

```

>>> template = scoretemplatetools.StringOrchestraScoreTemplate(
...     violin_count=6,

```

```

...     viola_count=4,
...     cello_count=3,
...     contrabass_count=2,
...     )
>>> score = template()

```

```

>>> score
Score="String Orchestra Score"<<4>>

```

Returns score template.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`StringOrchestraScoreTemplate.cello_count`

`StringOrchestraScoreTemplate.contrabass_count`

`StringOrchestraScoreTemplate.viola_count`

`StringOrchestraScoreTemplate.violin_count`

## Special methods

`StringOrchestraScoreTemplate.__call__()`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

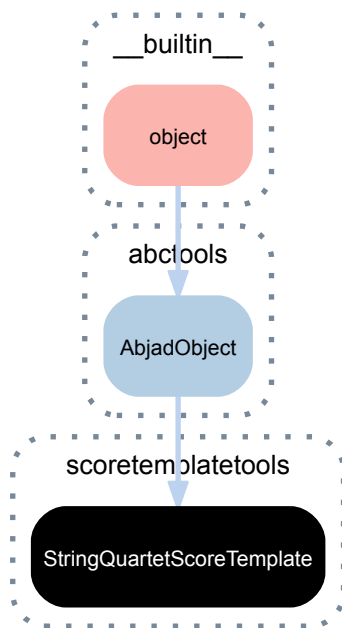
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 26.1.4 scoretemplatetools.StringQuartetScoreTemplate



**class** `scoretemplatetools.StringQuartetScoreTemplate`  
String quartet score template.

```
>>> template = scoretemplatetools.StringQuartetScoreTemplate()
>>> score = template()
```

```
>>> score
Score-"String Quartet Score"<<1>>
```

Returns score template.

### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

### Special methods

`StringQuartetScoreTemplate.__call__()`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

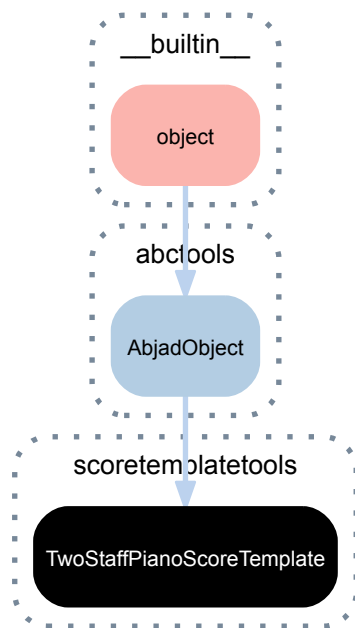
`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.



### 26.1.5 scoretemplatetools.TwoStaffPianoScoreTemplate



**class** `scoretemplatetools.TwoStaffPianoScoreTemplate`  
Two-staff piano score template.

```
>>> template = scoretemplatetools.TwoStaffPianoScoreTemplate()
>>> score = template()
```

```
>>> score
Score="Two-Staff Piano Score"<<1>>
```

Returns score template.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`TwoStaffPianoScoreTemplate.__call__()`

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

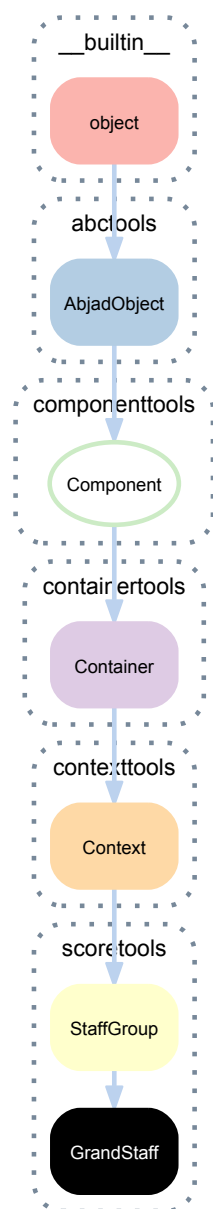
`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.



# SCORETOOLS

## 27.1 Concrete classes

### 27.1.1 scoretools.GrandStaff



**class** `scoretools.GrandStaff` (*music*, *context\_name*='GrandStaff', *name*=None)  
 Abjad model of grand staff:

```
>>> staff_1 = Staff("c'4 d'4 e'4 f'4 g'1")
>>> staff_2 = Staff("g2 f2 e1")

>>> grand_staff = scoretools.GrandStaff([staff_1, staff_2])
```

Returns grand staff.

## Bases

- `scoretools.StaffGroup`
- `contexttools.Context`
- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Context) **.engraver\_consists**

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

(Context) **.engraver\_removals**

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

(Context) **.is\_semantic**

(Context) **.lilypond\_format**

(Component) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Component) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

(Component).**storage\_format**

Storage format of component.

Returns string.

## Read/write properties

(Context).**context\_name**

Read / write name of context as a string.

(Context).**is\_nonsemantic**

Set indicator of nonsemantic voice:

```
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Returns boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interaction and other functions.

(Container).**is\_simultaneous**

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

(Context) **.name**

Read-write name of context. Must be string or none.

## Methods

(Container) **.append** (*component*)

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) **.extend** (*expr*)

Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```

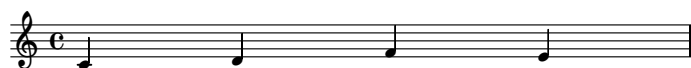


Returns none.

(Container) **.index** (*component*)

Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

(Container) **.pop** (*i=-1*)  
 Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

`(Container).remove(component)`

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

`(Container).reverse()`

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

`(Component).select(sequential=False)`

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

`(Container).select_leaves(start=0, stop=None, leaf_classes=None, recurse=True, allow_discontiguous_leaves=False)`

Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

`(Container).select_notes_and_chords()`

Selects notes and chords in container.



```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container) **.\_\_contains\_\_** (*expr*)

True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component) **.\_\_copy\_\_** (\*args)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(Container) **.\_\_delitem\_\_** (*i*)

Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Returns none.

(AbjadObject) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container) **.\_\_getitem\_\_** (*i*)

Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.

(Container) **.\_\_len\_\_** ()

Number of items in container.

Returns nonnegative integer.

(Component) **.\_\_mul\_\_** (*n*)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Context) **.\_\_repr\_\_** ()

(Component) **.\_\_rmul\_\_** (*n*)

Copies component *n* times and detach spanners.

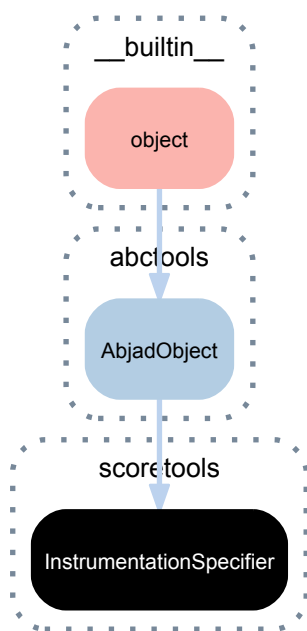
Returns list of new components.

(Container) **.\_\_setitem\_\_** (*i*, *expr*)

Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

Returns none.

## 27.1.2 scoretools.InstrumentationSpecifier



**class** `scoretools.InstrumentationSpecifier` (*performers=None*)

Abjad model of score instrumentation:

```
>>> flute = scoretools.Performer('Flute')
>>> flute.instruments.append(instrumenttools.Flute())
>>> flute.instruments.append(instrumenttools.AltoFlute())
```

```
>>> guitar = scoretools.Performer('Guitar')
>>> guitar.instruments.append(instrumenttools.Guitar())
```

```
>>> instrumentation_specifier = \
...     scoretools.InstrumentationSpecifier([flute, guitar])
```

```
>>> print instrumentation_specifier.storage_format
scoretools.InstrumentationSpecifier(
  performers=scoretools.PerformerInventory([
    scoretools.Performer(
      name='Flute',
      instruments=instrumenttools.InstrumentInventory([
        instrumenttools.Flute(),
        instrumenttools.AltoFlute()
      ])
    ),
    scoretools.Performer(
      name='Guitar',
      instruments=instrumenttools.InstrumentInventory([
        instrumenttools.Guitar()
      ])
    )
  ])
)
```

Returns instrumentation specifier.

### Bases

- `abctools.AbjadObject`
- `___builtin___object`

## Read-only properties

`InstrumentationSpecifier.instrument_count`  
Number of instruments in score:

```
>>> instrumentation_specifier.instrument_count
3
```

Returns nonnegative integer.

`InstrumentationSpecifier.instruments`  
List of instruments derived from performers:

```
>>> instrumentation_specifier.instruments
[Flute(), AltoFlute(), Guitar()]
```

Returns list.

`InstrumentationSpecifier.performer_count`  
Number of performers in score:

```
>>> instrumentation_specifier.performer_count
2
```

Returns nonnegative integer.

`InstrumentationSpecifier.performer_name_string`  
String of performer names:

```
>>> instrumentation_specifier.performer_name_string
'Flute, Guitar'
```

Returns string.

`InstrumentationSpecifier.storage_format`  
Storage format of instrumentation specifier.

Returns string.

## Read/write properties

`InstrumentationSpecifier.performers`  
Read / write list of performers in score:

```
>>> print instrumentation_specifier.performers.storage_format
scoretools.PerformerInventory([
    scoretools.Performer(
        name='Flute',
        instruments=instrumenttools.InstrumentInventory([
            instrumenttools.Flute(),
            instrumenttools.AltoFlute()
        ])
    ),
    scoretools.Performer(
        name='Guitar',
        instruments=instrumenttools.InstrumentInventory([
            instrumenttools.Guitar()
        ])
    )
])
```

Returns performer inventory.

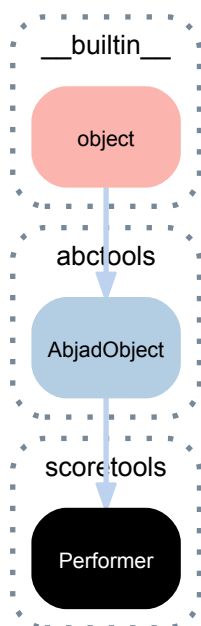
## Special methods

`InstrumentationSpecifier.__eq__(expr)`

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 27.1.3 scoretools.Performer



**class** scoretools.**Performer** (*name=None, instruments=None*)  
 A instrumental or vocal performer.

```
>>> performer = scoretools.Performer(name='flutist')
>>> performer.instruments.append(instrumenttools.Flute())
>>> performer.instruments.append(instrumenttools.Piccolo())
```

```
>>> print performer.storage_format
scoretools.Performer(
  name='flutist',
  instruments=instrumenttools.InstrumentInventory([
    instrumenttools.Flute(),
    instrumenttools.Piccolo()
  ])
)
```

The purpose of the class is to model things like flute I doubling piccolo and flute.

At present the class comprises an instrument inventory and name.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

**Performer.instrument\_count**

Number of instruments to be played by performer:

```
>>> performer.instrument_count
2
```

Returns nonnegative integer

**Performer.is\_doubling**

Is performer to play more than one instrument?

```
::
```

```
>>> performer.is_doubling
True
```

Returns boolean.

**Performer.likely\_instruments\_based\_on\_performer\_name**

Likely instruments based on performer name:

```
>>> for likely_instrument in \
...     performer.likely_instruments_based_on_performer_name:
...     likely_instrument.__name__
...
'AltoFlute'
'BassFlute'
'ContrabassFlute'
'Flute'
'Piccolo'
```

Returns list.

**Performer.most\_likely\_instrument\_based\_on\_performer\_name**

Most likely instrument based on performer name:

```
>>> performer.most_likely_instrument_based_on_performer_name
<class 'abjad.tools.instrumenttools.Flute.Flute.Flute'>
```

Returns instrument class.

**Performer.storage\_format**

Storage format of performer.

Returns string.

## Read/write properties

**Performer.instruments**

List of instruments to be played by performer:

```
>>> performer.instruments
InstrumentInventory([Flute(), Piccolo()])
```

Returns instrument inventory.

**Performer.name**

Score name of performer:

```
>>> performer.name
'flutist'
```

Returns string.

## Methods

`Performer.make_performer_name_instrument_dictionary()`

Make performer name / instrument dictionary:

```
>>> dictionary = \
...     performer.make_performer_name_instrument_dictionary()
>>> for key, value in sorted(dictionary.iteritems()):
...     print key + ':'
...     for x in value:
...         print '\t{}'.format(x.__name__)
accordionist:
    Accordion
baritone:
    BaritoneVoice
bass:
    BassVoice
bassist:
    Contrabass
bassoonist:
    Bassoon
    Contrabassoon
brass player:
    AltoTrombone
    BassTrombone
    FrenchHorn
    TenorTrombone
    Trumpet
    Tuba
cellist:
    Cello
clarinetist:
    BassClarinet
    BFlatClarinet
    ClarinetInA
    ContrabassClarinet
    EFlatClarinet
clarinettist:
    BassClarinet
    BFlatClarinet
    ClarinetInA
    ContrabassClarinet
    EFlatClarinet
contrabassist:
    Contrabass
contralto:
    ContraltoVoice
double reed player:
    Bassoon
    Contrabassoon
    EnglishHorn
    Oboe
flautist:
    AltoFlute
    BassFlute
    ContrabassFlute
    Flute
    Piccolo
flutist:
    AltoFlute
    BassFlute
    ContrabassFlute
    Flute
    Piccolo
guitarist:
    Guitar
harpist:
    Harp
harpsichordist:
    Harpsichord
hornist:
    FrenchHorn
```

```

instrumentalist:
    Accordion
    AltoFlute
    AltoSaxophone
    AltoTrombone
    BaritoneSaxophone
    BaritoneVoice
    BassClarinet
    BassFlute
    Bassoon
    BassSaxophone
    BassTrombone
    BassVoice
    BFlatClarinet
    Cello
    ClarinetInA
    Contrabass
    ContrabassClarinet
    ContrabassFlute
    Contrabassoon
    ContrabassSaxophone
    ContraltoVoice
    EFlatClarinet
    EnglishHorn
    Flute
    FrenchHorn
    Glockenspiel
    Guitar
    Harp
    Harpsichord
    Marimba
    MezzoSopranoVoice
    Oboe
    Piano
    Piccolo
    SopraninoSaxophone
    SopranoSaxophone
    SopranoVoice
    TenorSaxophone
    TenorTrombone
    TenorVoice
    Trumpet
    Tuba
    UntunedPercussion
    Vibraphone
    Viola
    Violin
    Xylophone
keyboardist:
    Accordion
    Harpsichord
    Piano
mezzo-soprano:
    MezzoSopranoVoice
oboist:
    EnglishHorn
    Oboe
percussionist:
    Glockenspiel
    Marimba
    UntunedPercussion
    Vibraphone
    Xylophone
pianist:
    Piano
reed player:
    AltoSaxophone
    BaritoneSaxophone
    BassClarinet
    Bassoon
    BassSaxophone
    BFlatClarinet
    ClarinetInA

```

```

    ContrabassClarinet
    Contrabassoon
    ContrabassSaxophone
    EFlatClarinet
    EnglishHorn
    Oboe
    SopraninoSaxophone
    SopranoSaxophone
    TenorSaxophone
saxophonist:
    AltoSaxophone
    BaritoneSaxophone
    BassSaxophone
    ContrabassSaxophone
    SopraninoSaxophone
    SopranoSaxophone
    TenorSaxophone
single reed player:
    AltoSaxophone
    BaritoneSaxophone
    BassClarinet
    BassSaxophone
    BFlatClarinet
    ClarinetInA
    ContrabassClarinet
    ContrabassSaxophone
    EFlatClarinet
    SopraninoSaxophone
    SopranoSaxophone
    TenorSaxophone
soprano:
    SopranoVoice
string player:
    Cello
    Contrabass
    Guitar
    Harp
    Viola
    Violin
tenor:
    TenorVoice
trombonist:
    AltoTrombone
    BassTrombone
    TenorTrombone
trumpeter:
    Trumpet
tubist:
    Tuba
vibraphonist:
    Vibraphone
violinist:
    Violin
violist:
    Viola
vocalist:
    BaritoneVoice
    BassVoice
    ContraltoVoice
    MezzoSopranoVoice
    SopranoVoice
    TenorVoice
wind player:
    AltoFlute
    ...
    TenorSaxophone
xylophonist:
    Xylophone

```

Returns ordered dictionary.



## Static methods

`Performer.list_performer_names()`

Lists performer names.

```
>>> for name in scoretools.Performer.list_performer_names():
...     name
...
'accordionist'
'baritone'
'bass'
'bassist'
'bassoonist'
'cellist'
'clarinetist'
'contralto'
'flutist'
'guitarist'
'harpist'
'harpsichordist'
'hornist'
'mezzo-soprano'
'oboist'
'percussionist'
'pianist'
'saxophonist'
'soprano'
'tenor'
'trombonist'
'trumpeter'
'tubist'
'vibraphonist'
'violinist'
'violist'
'xylophonist'
```

Returns list.

`Performer.list_primary_performer_names()`

List performer names:

```
>>> for pair in scoretools.Performer.list_primary_performer_names():
...     pair
...
('accordionist', 'acc.')
('baritone', 'bar.')
('bass', 'bass')
('bassist', 'vb.')
('bassoonist', 'bsn.')
('cellist', 'vc.')
('clarinetist', 'cl.')
('contralto', 'contr.')
('flutist', 'fl.')
('guitarist', 'gt.')
('harpist', 'hp.')
('harpsichordist', 'hpschd.')
('hornist', 'hn.')
('mezzo-soprano', 'ms.')
('oboist', 'ob.')
('pianist', 'pf.')
('saxophonist', 'alt. sax.')
('soprano', 'sop.')
('tenor', 'ten.')
('trombonist', 'trb.')
('trumpeter', 'tp.')
('tubist', 'tb.')
('violinist', 'vn.')
('violist', 'va.')

```

Returns list.

## Special methods

`Performer.__eq__(expr)`

`Performer.__hash__()`

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

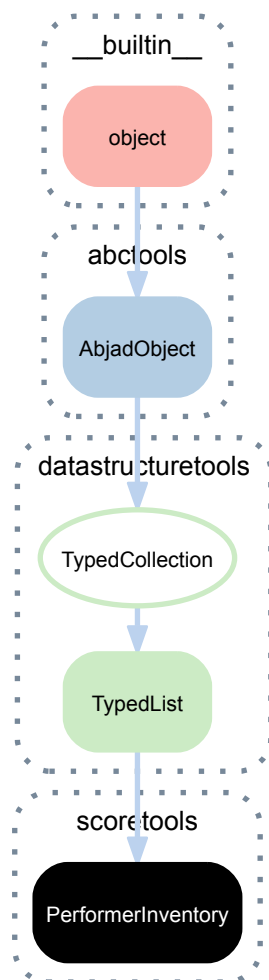
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 27.1.4 scoretools.PerformerInventory



**class** `scoretools.PerformerInventory` (*tokens=None, item\_class=None, name=None*)

Abjad model of an ordered list of performers.

Performer inventories implement the list interface and are mutable.

## Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedList).append(token)`  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

`(TypedList).count(token)`  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)
3
```

Returns count.

`(TypedList).extend(tokens)`  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

`(TypedList).index(token)`  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList (
...     tokens=('c'qf', "as'", 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

(TypedList) **.insert** (*i*, *token*)  
 Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('1', 2, 4.3))
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None*, *item\_class=None*, *name=None*)

(TypedList) **.pop** (*i=-1*)  
 Aliases list.pop().

(TypedList) **.remove** (*token*)  
 Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()  
 Aliases list.reverse().

(TypedList) **.sort** (*cmp=None*, *key=None*, *reverse=False*)  
 Aliases list.sort().

## Special methods

(TypedCollection) **.\_\_contains\_\_** (*token*)

(TypedList) **.\_\_delitem\_\_** (*i*)  
 Aliases list.\_\_delitem\_\_().

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedList) **.\_\_getitem\_\_** (*i*)  
 Aliases list.\_\_getitem\_\_().

(TypedList) **.\_\_iadd\_\_** (*expr*)  
 Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'fff',
        target_context=stafftools.Staff
    )
],
item_class=contexttools.DynamicMark
)
```

Returns collection.

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

(TypedCollection).**\_\_ne\_\_**(*expr*)

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList).**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList).**\_\_setitem\_\_**(*i*, *expr*)

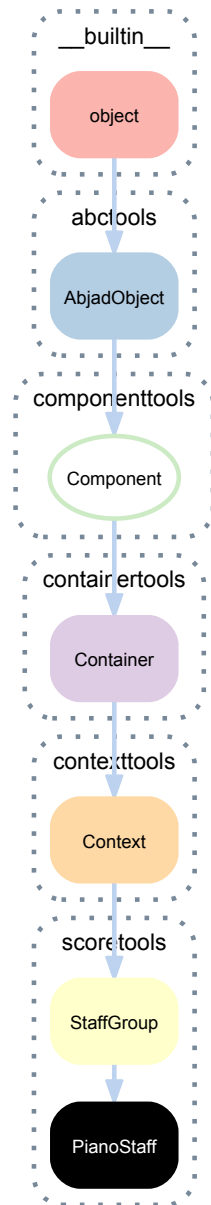
Change tokens in *expr* to items and set:

```
>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c"),
    pitchtools.NamedPitch("d"),
    pitchtools.NamedPitch("e"),
    pitchtools.NamedPitch('gqs,')
],
item_class=pitchtools.NamedPitch
)
```

```
>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c"),
    pitchtools.NamedPitch("d"),
    pitchtools.NamedPitch("e"),
    pitchtools.NamedPitch("f"),
    pitchtools.NamedPitch("g"),
    pitchtools.NamedPitch("a"),
    pitchtools.NamedPitch("b"),
    pitchtools.NamedPitch("c'")
])
```

```
],
    item_class=pitchtools.NamedPitch
)
```

### 27.1.5 scoretools.PianoStaff



**class** `scoretools.PianoStaff` (*music=None, context\_name='PianoStaff', name=None*)  
 Abjad model of piano staff:

```
>>> staff_1 = Staff("c'4 d'4 e'4 f'4 g'1")
>>> staff_2 = Staff("g2 f2 e1")
```

```
>>> piano_staff = scoretools.PianoStaff([staff_1, staff_2])
```

Returns piano staff.

#### Bases

- `scoretools.StaffGroup`

- `contexttools.Context`
- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `(Context).engraver_consists`

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

### `(Context).engraver_removals`

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

### `(Context).is_semantic`

### `(Context).lilypond_format`

### `(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

### `(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

### `(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

### `(Context).context_name`

Read / write name of context as a string.

### `(Context).is_nonsemantic`

Set indicator of nonsemantic voice:

```
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Returns boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interaction and other functions.

(Container).**.is\_simultaneous**

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

(Context).**.name**

Read-write name of context. Must be string or none.



## Methods

(Container) .**append**(*component*)  
 Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) .**extend**(*expr*)  
 Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



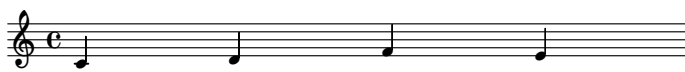
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) .**index**(*component*)  
 Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) .**insert**(*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
```

```
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2. Insert note. Fracture spanners:**

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

`(Container).pop(i=-1)`

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

`(Container).remove(component)`

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse()**  
Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (*sequential=False*)  
Selects component.  
  
Returns component selection when *sequential* is false.  
  
Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)  
Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords()**  
Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container) **.\_\_contains\_\_** (*expr*)  
True when *expr* appears in container. Otherwise false.  
  
Returns boolean.

(Component) **.\_\_copy\_\_** (*\*args*)  
Copies component with marks but without children of component or spanners attached to component.  
  
Returns new component.

(Container) .**\_\_delitem\_\_**(*i*)  
 Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners.  
 Preserve spanners that component(s) cover(s).  
 Returns none.

(AbjadObject) .**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(Container) .**\_\_getitem\_\_**(*i*)  
 Get container *i*. Shallow traversal of container for numeric indices only.  
 Returns component.

(Container) .**\_\_len\_\_**()  
 Number of items in container.  
 Returns nonnegative integer.

(Component) .**\_\_mul\_\_**(*n*)  
 Copies component *n* times and detaches spanners.  
 Returns list of new components.

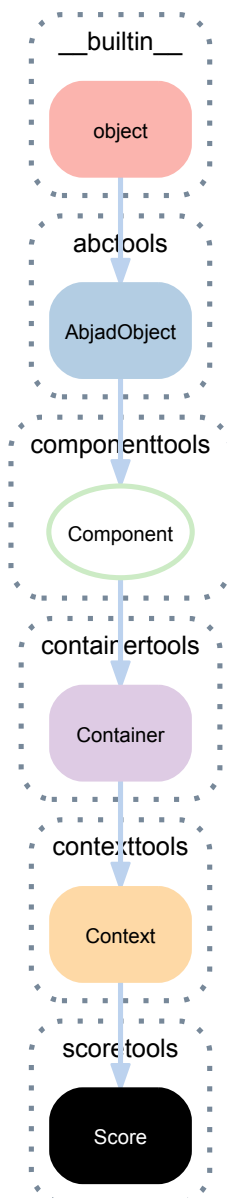
(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Context) .**\_\_repr\_\_**()

(Component) .**\_\_rmul\_\_**(*n*)  
 Copies component *n* times and detach spanners.  
 Returns list of new components.

(Container) .**\_\_setitem\_\_**(*i, expr*)  
 Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with '*expr*'. Reattach spanners to new contents. This operation always leaves score tree in tact.  
 Returns none.

### 27.1.6 scoretools.Score



**class** `scoretools.Score` (*music=None*, *context\_name='Score'*, *name=None*)  
 A score.

```

>>> staff_1 = Staff("c'8 d'8 e'8 f'8")
>>> staff_2 = Staff("c'8 d'8 e'8 f'8")
>>> score = Score([staff_1, staff_2])
>>> show(score)
  
```



#### Bases

- `contexttools.Context`
- `containertools.Container`

- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `(Context).engraver_consists`

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

### `(Context).engraver_removals`

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

### `(Context).is_semantic`

### `(Context).lilypond_format`

### `(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

### `(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

### `(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

### `(Context).context_name`

Read / write name of context as a string.

### `(Context).is_nonsemantic`

Set indicator of nonsemantic voice:

```
>>> measures = \
...   measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Returns boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interaction and other functions.

(Container).**.is\_simultaneous**  
Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

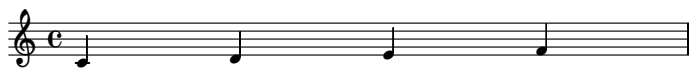
(Context).**.name**  
Read-write name of context. Must be string or none.

## Methods

Score.**.add\_double\_bar()**  
Add double bar to end of score.

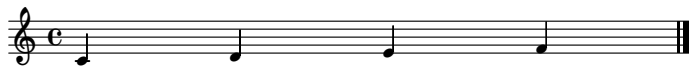
```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> score = Score([staff])
```

```
>>> show(score)
```



```
>>> score.add_double_bar()
BarLine('|.|')(f'4)
```

```
>>> show(score)
```



Returns bar line.

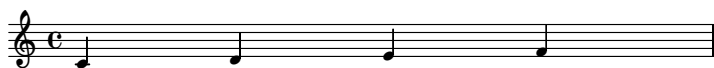
`Score.add_final_markup(markup, extra_offset=None)`

Add *markup* to end of score:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> score = Score([staff])
>>> markup = r'\italic \right-column { "Bremen - Boston - LA." "Jul 2010 - May 2011." }'
>>> markup = markuptools.Markup(markup, Down)
>>> markup = score.add_final_markup(markup, extra_offset=(4, -2))
```

```
>>> print markup.storage_format
markuptools.Markup((
    markuptools.MarkupCommand(
        'italic',
        markuptools.MarkupCommand(
            'right-column',
            [
                'Bremen - Boston - LA.',
                'Jul 2010 - May 2011.'
            ]
        )
    ),
    direction=Down
))
```

```
>>> show(staff)
```



*Bremen - Boston - LA.*  
*Jul 2010 - May 2011.*

Return *markup*.

`(Container).append(component)`

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.



(Container) .**extend**(*expr*)  
 Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) .**index**(*component*)  
 Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) .**insert**(*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
```

```
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

(Container) **.pop** (*i*=-1)

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

(Container) **.remove** (*component*)

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse** ()

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (*sequential=False*)  
Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)  
Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest("r8"), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords** ()  
Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container) **.\_\_contains\_\_** (*expr*)  
True when *expr* appears in container. Otherwise false.  
Returns boolean.

(Component) **.\_\_copy\_\_** (\*args)  
Copies component with marks but without children of component or spanners attached to component.  
Returns new component.

(Container) **.\_\_delitem\_\_** (*i*)  
Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).  
Returns none.

(AbjadObject) **.\_\_eq\_\_** (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(Container) **.\_\_getitem\_\_** (*i*)  
Get container *i*. Shallow traversal of container for numeric indices only.  
Returns component.

(Container) **.\_\_len\_\_** ()  
Number of items in container.  
Returns nonnegative integer.

(Component) .\_\_mul\_\_(*n*)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject) .\_\_ne\_\_(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Context) .\_\_repr\_\_()

(Component) .\_\_rmul\_\_(*n*)

Copies component *n* times and detach spanners.

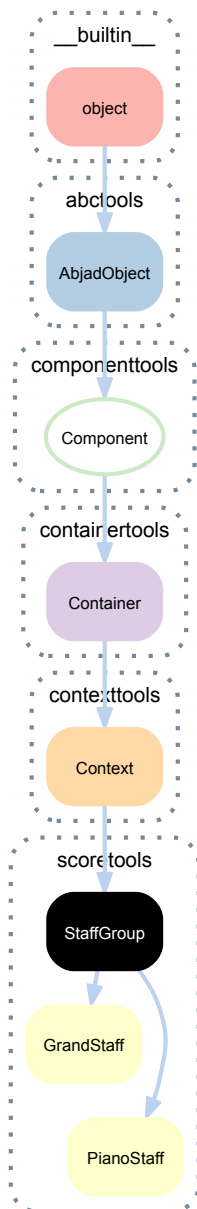
Returns list of new components.

(Container) .\_\_setitem\_\_(*i*, *expr*)

Set container *i* equal to *expr*. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

Returns none.

### 27.1.7 scoretools.StaffGroup



**class** `scoretools.StaffGroup` (*music=None, context\_name='StaffGroup', name=None*)  
 Abjad model of staff group:

```
>>> staff_1 = Staff("c'4 d'4 e'4 f'4 g'1")
>>> staff_2 = Staff("g2 f2 e1")
```

```
>>> staff_group = scoretools.StaffGroup([staff_1, staff_2])
```

Returns staff group.

#### Bases

- `contexttools.Context`
- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`

- `__builtin__.object`

## Read-only properties

### `(Context).engraver_consists`

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

### `(Context).engraver_removals`

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

### `(Context).is_semantic`

### `(Context).lilypond_format`

### `(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

### `(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

### `(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

### `(Context).context_name`

Read / write name of context as a string.

### `(Context).is_nonsemantic`

Set indicator of nonsemantic voice:

```
>>> measures = \
...   measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Returns boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice iteration and other functions.

(Container) **.is\_simultaneous**

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

(Context) **.name**

Read-write name of context. Must be string or none.

## Methods

(Container) **.append** (*component*)

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) **.extend** (*expr*)  
Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



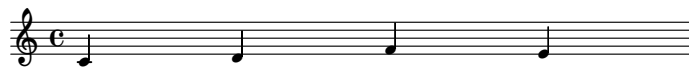
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) **.index** (*component*)  
Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```





Returns none.

(Container) **.reverse()**  
Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select(sequential=False)**  
Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves(start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False)**  
Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords()**  
Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container) **.\_\_contains\_\_(expr)**  
True when *expr* appears in container. Otherwise false.  
Returns boolean.

(Component) **.\_\_copy\_\_(\*args)**  
Copies component with marks but without children of component or spanners attached to component.  
Returns new component.

(Container) **.\_\_delitem\_\_(i)**  
Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).  
Returns none.

(AbjadObject) **.\_\_eq\_\_(expr)**  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(Container).**\_\_getitem\_\_**(*i*)  
Get container *i*. Shallow traversal of container for numeric indices only.  
Returns component.

(Container).**\_\_len\_\_**()  
Number of items in container.  
Returns nonnegative integer.

(Component).**\_\_mul\_\_**(*n*)  
Copies component *n* times and detaches spanners.  
Returns list of new components.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(Context).**\_\_repr\_\_**()

(Component).**\_\_rmul\_\_**(*n*)  
Copies component *n* times and detach spanners.  
Returns list of new components.

(Container).**\_\_setitem\_\_**(*i*, *expr*)  
Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with '*expr*'. Reattach spanners to new contents. This operation always leaves score tree in tact.  
Returns none.

## 27.2 Functions

### 27.2.1 scoretools.make\_empty\_piano\_score

scoretools.**make\_empty\_piano\_score**()  
Make empty piano score:

```
>>> score, treble, bass = scoretools.make_empty_piano_score()
```

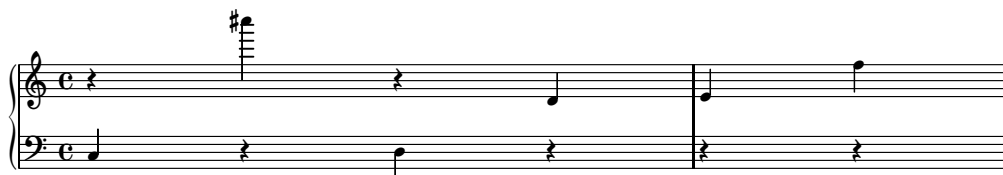
Returns score, treble staff, bass staff.

### 27.2.2 scoretools.make\_piano\_score\_from\_leaves

scoretools.**make\_piano\_score\_from\_leaves**(*leaves*, *lowest\_treble\_pitch=None*)  
Make piano score from *leaves*:

```
>>> notes = [Note(x, (1, 4)) for x in [-12, 37, -10, 2, 4, 17]]
>>> score, treble_staff, bass_staff = scoretools.make_piano_score_from_leaves(notes)
```

```
>>> show(score)
```



When *lowest\_treble\_pitch=None* set to B3.

Returns score, treble staff, bass staff.

### 27.2.3 scoretools.make\_piano\_sketch\_score\_from\_leaves

`scoretools.make_piano_sketch_score_from_leaves` (*leaves*, *lowest\_treble\_pitch=None*)

Make piano sketch score from *leaves*:

```
>>> notes = notetools.make_notes([-12, -10, -8, -7, -5, 0, 2, 4, 5, 7], [(1, 4)])
>>> score, treble_staff, bass_staff = scoretools.make_piano_sketch_score_from_leaves(notes)
```

```
>>> show(score)
```



When `lowest_treble_pitch=None` set to B3.

Make time signatures and bar numbers transparent.

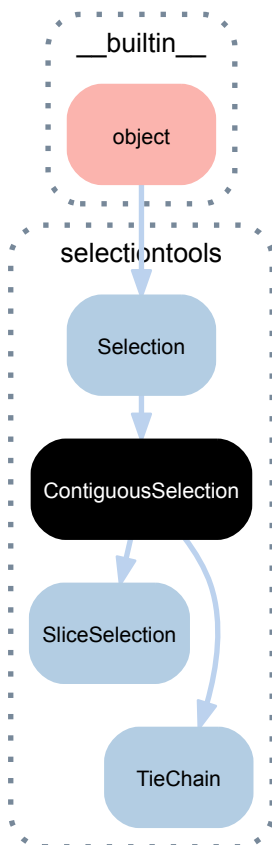
Do not print bar lines or span bars.

Returns score, treble staff, bass staff.

# SELECTIONTOOLS

## 28.1 Concrete classes

### 28.1.1 selectiontools.ContiguousSelection



**class** `selectiontools.ContiguousSelection` (*music=None*)  
A time-contiguous selection of components.

#### Bases

- `selectiontools.Selection`
- `__builtin__.object`

## Methods

`(Selection).get_duration(in_seconds=False)`

Gets duration of contiguous selection.

Returns duration.

`(Selection).get_spanners(spanner_classes=None)`

Gets spanners attached to any component in selection.

Returns set.

`ContiguousSelection.get_timespan(in_seconds=False)`

Gets timespan of contiguous selection.

Returns timespan.

`ContiguousSelection.group_by(predicate)`

Groups components in contiguous selection by *predicate*.

Returns list of tuples.

`ContiguousSelection.partition_by_durations(durations, cyclic=False, fill='exact', in_seconds=False, overhang=False)`

Partitions *components* according to *durations*.

When *fill* is 'exact' then parts must equal *durations* exactly.

When *fill* is 'less' then parts must be less than or equal to *durations*.

When *fill* is 'greater' then parts must be greater or equal to *durations*.

Reads *durations* cyclically when *cyclic* is true.

Reads component durations in seconds when *in\_seconds* is true.

Returns remaining components at end in final part when *overhang* is true.

`ContiguousSelection.partition_by_durations_exactly(durations, cyclic=False, in_seconds=False, overhang=False)`

`ContiguousSelection.partition_by_durations_not_greater_than(durations, cyclic=False, in_seconds=False, overhang=False)`

`ContiguousSelection.partition_by_durations_not_less_than(durations, cyclic=False, in_seconds=False, overhang=False)`

## Special methods

`ContiguousSelection.__add__(expr)`

Add *expr* to slice selection.

Returns new slice selection.

`(Selection).__contains__(expr)`

True when *expr* is in selection. Otherwise false.

Returns boolean.

`(Selection).__eq__(expr)`

True when selection and *expr* are of the same type and when music of selection equals music of *expr*. Otherwise false.

Returns boolean.

(Selection).**\_\_getitem\_\_**(*expr*)  
Gets item *expr* from selection.

Returns component from selection.

(Selection).**\_\_len\_\_**()  
Number of components in selection.

Returns nonnegative integer.

(Selection).**\_\_ne\_\_**(*expr*)  
True when selection does not equal *expr*. Otherwise false.

Returns boolean.

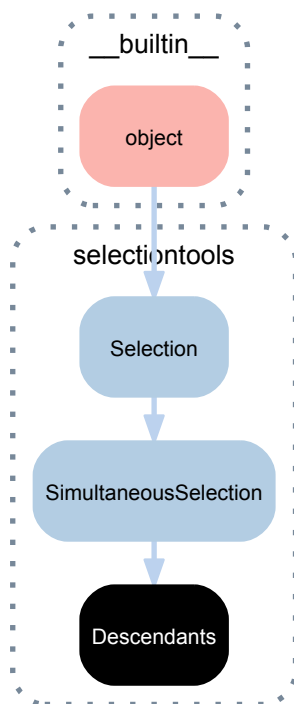
ContiguousSelection.**\_\_radd\_\_**(*expr*)  
Add slice selection to *expr*.

Returns new slice selection.

(Selection).**\_\_repr\_\_**()  
Interpreter representation of selection.

Returns string.

## 28.1.2 selectiontools.Descendants



**class** selectiontools.**Descendants** (*component=None, cross\_offset=None, include\_self=True*)  
Abjad model of Component descendants:

```
>>> score = Score()
>>> score.append(Staff(r"""\new Voice = "Treble Voice" { c'4 }""",
...   name='Treble Staff'))
>>> score.append(Staff(r"""\new Voice = "Bass Voice" { b,4 }""",
...   name='Bass Staff'))
```

```
>>> for x in selectiontools.Descendants(score): x
...
Score<<2>>
```

```
Staff="Treble Staff"{1}
Voice="Treble Voice"{1}
Note("c'4")
Staff="Bass Staff"{1}
Voice="Bass Voice"{1}
Note('b,4')
```

```
>>> for x in selectiontools.Descendants(score['Bass Voice']): x
...
Voice="Bass Voice"{1}
Note('b,4')
```

Descendants is treated as the selection of the component's improper descendants.

Returns descendants instance.

## Bases

- `selectiontools.SimultaneousSelection`
- `selectiontools.Selection`
- `__builtin__.object`

## Read-only properties

`Descendants.component`

The component from which the selection was derived.

## Methods

`(Selection).get_duration(in_seconds=False)`

Gets duration of contiguous selection.

Returns duration.

`(Selection).get_spanners(spanner_classes=None)`

Gets spanners attached to any component in selection.

Returns set.

`(SimultaneousSelection).get_vertical_moment_at(offset)`

Select vertical moment at *offset*.

## Special methods

`(Selection).__add__(expr)`

Cocatenates *expr* to selection.

Returns new selection.

`(Selection).__contains__(expr)`

True when *expr* is in selection. Otherwise false.

Returns boolean.

`(Selection).__eq__(expr)`

True when selection and *expr* are of the same type and when music of selection equals music of *expr*. Otherwise false.

Returns boolean.



(Selection).**\_\_getitem\_\_**(*expr*)  
 Gets item *expr* from selection.  
 Returns component from selection.

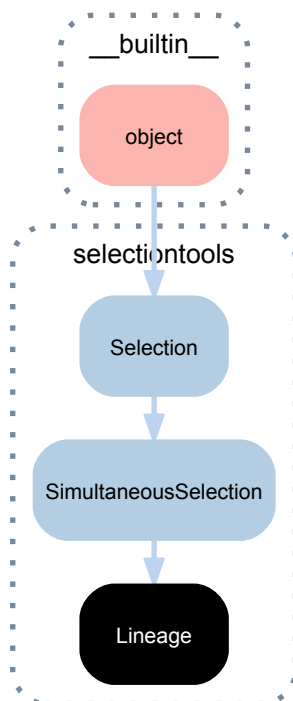
(Selection).**\_\_len\_\_**()  
 Number of components in selection.  
 Returns nonnegative integer.

(Selection).**\_\_ne\_\_**(*expr*)  
 True when selection does not equal *expr*. Otherwise false.  
 Returns boolean.

(Selection).**\_\_radd\_\_**(*expr*)  
 Concatenates selection to *expr*.  
 Returns newly created selection.

(Selection).**\_\_repr\_\_**()  
 Interpreter representation of selection.  
 Returns string.

### 28.1.3 selectiontools.Lineage



**class** selectiontools.**Lineage** (*component*)  
 Abjad model of Component lineage:

```

>>> score = Score()
>>> score.append(Staff(r"""\new Voice = "Treble Voice" { c'4 }""",
... name='Treble Staff'))
>>> score.append(Staff(r"""\new Voice = "Bass Voice" { b,4 }""",
... name='Bass Staff'))
  
```

```

>>> for x in selectiontools.Lineage(score): x
...
Score<<2>>
Staff-"Treble Staff"{1}
Voice-"Treble Voice"{1}
  
```

```
Note("c'4")
Staff="Bass Staff"{1}
Voice="Bass Voice"{1}
Note('b,4')
```

```
>>> for x in selectiontools.Lineage(score['Bass Voice']): x
...
Score<<2>>
Staff="Bass Staff"{1}
Voice="Bass Voice"{1}
Note('b,4')
```

Returns lineage instance.

## Bases

- `selectiontools.SimultaneousSelection`
- `selectiontools.Selection`
- `__builtin__.object`

## Read-only properties

`Lineage.component`

The component from which the selection was derived.

## Methods

`(Selection).get_duration(in_seconds=False)`

Gets duration of contiguous selection.

Returns duration.

`(Selection).get_spanners(spanner_classes=None)`

Gets spanners attached to any component in selection.

Returns set.

`(SimultaneousSelection).get_vertical_moment_at(offset)`

Select vertical moment at *offset*.

## Special methods

`(Selection).__add__(expr)`

Cocatenates *expr* to selection.

Returns new selection.

`(Selection).__contains__(expr)`

True when *expr* is in selection. Otherwise false.

Returns boolean.

`(Selection).__eq__(expr)`

True when selection and *expr* are of the same type and when music of selection equals music of *expr*. Otherwise false.

Returns boolean.

`(Selection).__getitem__(expr)`

Gets item *expr* from selection.

Returns component from selection.

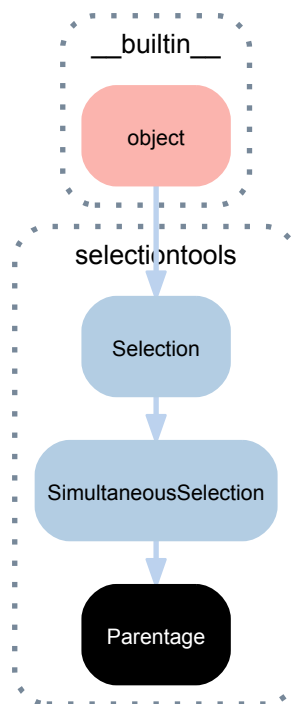
(Selection). **\_\_len\_\_**()  
 Number of components in selection.  
 Returns nonnegative integer.

(Selection). **\_\_ne\_\_**(*expr*)  
 True when selection does not equal *expr*. Otherwise false.  
 Returns boolean.

(Selection). **\_\_radd\_\_**(*expr*)  
 Concatenates selection to *expr*.  
 Returns newly created selection.

(Selection). **\_\_repr\_\_**()  
 Interpreter representation of selection.  
 Returns string.

### 28.1.4 selectiontools.Parentage



**class** selectiontools.**Parentage** (*component=None, include\_self=True*)  
 The parentage of a component.

```

>>> score = Score()
>>> string = r"""\new Voice = "Treble Voice" { e'4 }""
>>> treble_staff = Staff(string, name='Treble Staff')
>>> score.append(treble_staff)
>>> string = r"""\new Voice = "Bass Voice" { c4 }""
>>> bass_staff = Staff(string, name='Bass Staff')
>>> clef = contexttools.ClefMark('bass')
>>> clef = clef.attach(bass_staff)
>>> score.append(bass_staff)
>>> show(score)
  
```



```
>>> bass_voice = score['Bass Voice']
>>> note = bass_voice[0]
>>> parentage = inspect(note).get_parentage()
```

```
>>> for x in parentage: x
...
Note('c4')
Voice-"Bass Voice"{1}
Staff-"Bass Staff"{1}
Score<<2>>
```

## Bases

- `selectiontools.SimultaneousSelection`
- `selectiontools.Selection`
- `__builtin__.object`

## Read-only properties

### `Parentage.component`

The component from which the selection was derived.

Returns component.

### `Parentage.depth`

Length of proper parentage of component.

Returns nonnegative integer.

### `Parentage.is_orphan`

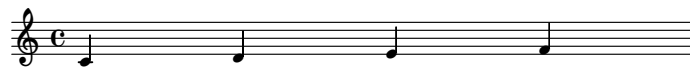
True when component has no parent. Otherwise false.

Returns boolean.

### `Parentage.logical_voice_indicator`

Logical voice indicator of component.

```
>>> voice = Voice("c'4 d'4 e'4 f'4", name='CustomVoice')
>>> staff = Staff([voice], name='CustomStaff')
>>> score = Score([staff], name='CustomScore')
>>> show(score)
```



```
>>> leaf = score.select_leaves()[0]
>>> parentage = inspect(leaf).get_parentage()
>>> indicator = parentage.logical_voice_indicator
```

```
>>> for key, value in indicator.iteritems():
...     print '%12s: %s' % (key, value)
...
      score: Score-'CustomScore'
  staff group:
    staff: Staff-'CustomStaff'
    voice: Voice-'CustomVoice'
```

Returns ordered dictionary.

### `Parentage.parent`

Parent of component.

Returns none when component has no parent.

Returns component or none.

`Parentage.prolation`

Prolation governing component.

Returns multiplier.

`Parentage.root`

Last element in parentage.

Returns component.

`Parentage.score_index`

Score index of component.

```
>>> staff_1 = Staff(r"\times 2/3 { c'2 b'2 a'2 }")
>>> staff_2 = Staff("c'2 d'2")
>>> score = Score([staff_1, staff_2])
>>> show(score)
```



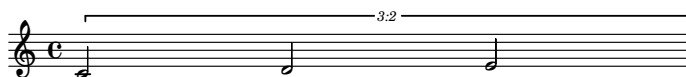
```
>>> leaves = score.select_leaves(allow_discontiguous_leaves=True)
>>> for leaf in leaves:
...     parentage = inspect(leaf).get_parentage()
...     leaf, parentage.score_index
...
(Note("c'2"), (0, 0, 0))
(Note("b'2"), (0, 0, 1))
(Note("a'2"), (0, 0, 2))
(Note("c'2"), (1, 0))
(Note("d'2"), (1, 1))
```

Returns tuple of zero or more nonnegative integers.

`Parentage.tuplet_depth`

Tuplet depth of component.

```
>>> tuplet = Tuplet(Multiplier(2, 3), "c'2 d'2 e'2")
>>> staff = Staff([tuplet])
>>> note = staff.select_leaves()[0]
>>> show(staff)
```



```
>>> inspect(note).get_parentage().tuplet_depth
1
```

```
>>> inspect(tuplet).get_parentage().tuplet_depth
0
```

```
>>> inspect(staff).get_parentage().tuplet_depth
0
```

Returns nonnegative integer.

## Methods

`(Selection).get_duration(in_seconds=False)`

Gets duration of contiguous selection.

Returns duration.

`Parentage.get_first(component_classes=None)`

Gets first instance of *component\_classes* in parentage.

Returns component or none.

(Selection) **.get\_spanners** (*spanner\_classes=None*)  
Gets spanners attached to any component in selection.

Returns set.

(SimultaneousSelection) **.get\_vertical\_moment\_at** (*offset*)  
Select vertical moment at *offset*.

## Special methods

(Selection) **.\_\_add\_\_** (*expr*)  
Cocatenates *expr* to selection.

Returns new selection.

(Selection) **.\_\_contains\_\_** (*expr*)  
True when *expr* is in selection. Otherwise false.

Returns boolean.

(Selection) **.\_\_eq\_\_** (*expr*)  
True when selection and *expr* are of the same type and when music of selection equals music of *expr*.  
Otherwise false.

Returns boolean.

(Selection) **.\_\_getitem\_\_** (*expr*)  
Gets item *expr* from selection.

Returns component from selection.

(Selection) **.\_\_len\_\_** ()  
Number of components in selection.

Returns nonnegative integer.

(Selection) **.\_\_ne\_\_** (*expr*)  
True when selection does not equal *expr*. Otherwise false.

Returns boolean.

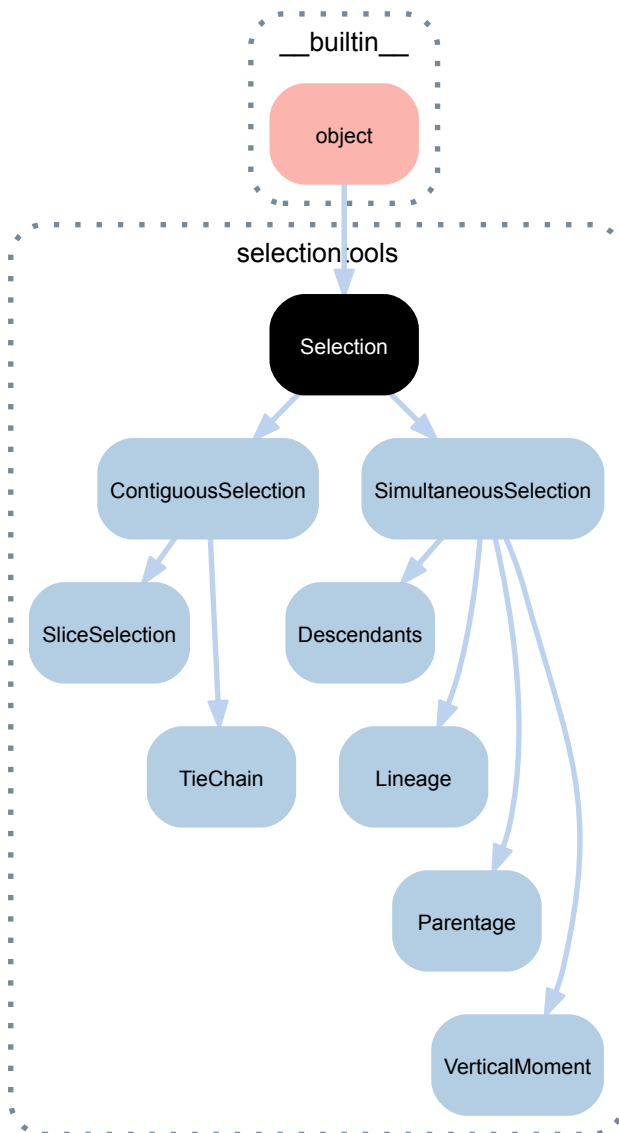
(Selection) **.\_\_radd\_\_** (*expr*)  
Concatenates selection to *expr*.

Returns newly created selection.

(Selection) **.\_\_repr\_\_** ()  
Interpreter representation of selection.

Returns string.

### 28.1.5 selectiontools.Selection



**class** `selectiontools.Selection` (*music=None*)  
 A selection of components.

#### Bases

- `__builtin__.object`

#### Methods

`Selection.get_duration` (*in\_seconds=False*)

Gets duration of contiguous selection.

Returns duration.

`Selection.get_spanners` (*spanner\_classes=None*)

Gets spanners attached to any component in selection.

Returns set.

## Special methods

`Selection.__add__(expr)`

Cocatenates *expr* to selection.

Returns new selection.

`Selection.__contains__(expr)`

True when *expr* is in selection. Otherwise false.

Returns boolean.

`Selection.__eq__(expr)`

True when selection and *expr* are of the same type and when music of selection equals music of *expr*. Otherwise false.

Returns boolean.

`Selection.__getitem__(expr)`

Gets item *expr* from selection.

Returns component from selection.

`Selection.__len__()`

Number of components in selection.

Returns nonnegative integer.

`Selection.__ne__(expr)`

True when selection does not equal *expr*. Otherwise false.

Returns boolean.

`Selection.__radd__(expr)`

Concatenates selection to *expr*.

Returns newly created selection.

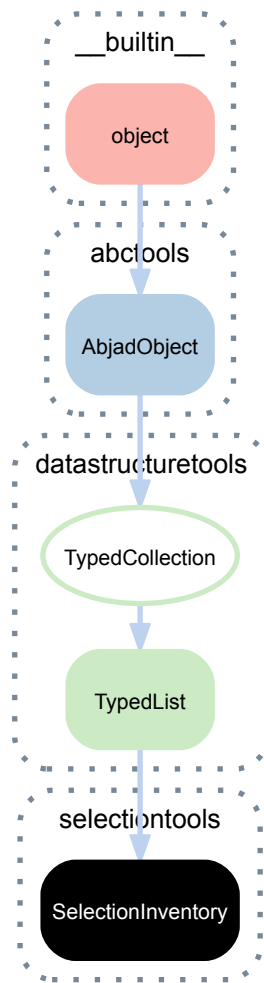
`Selection.__repr__()`

Interpreter representation of selection.

Returns string.



### 28.1.6 selectiontools.SelectionInventory



**class** selectiontools.**SelectionInventory** (*tokens=None, item\_class=None, name=None*)

An inventory of component selections:

```
>>> inventory = selectiontools.SelectionInventory()
```

#### Bases

- datastructuretools.TypedList
- datastructuretools.TypedCollection
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

(TypedCollection).**item\_class**

Item class to coerce tokens into.

(TypedCollection).**storage\_format**

Storage format of typed tuple.

## Read/write properties

(TypedCollection) .**name**  
Read / write name of typed tuple.

## Methods

(TypedList) .**append** (*token*)  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(  
...     item_class=int)  
>>> integer_collection[:]  
[]
```

```
>>> integer_collection.append('1')  
>>> integer_collection.append(2)  
>>> integer_collection.append(3.4)  
>>> integer_collection[:]  
[1, 2, 3]
```

Returns none.

(TypedList) .**count** (*token*)  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(  
...     tokens=[0, 0., '0', 99],  
...     item_class=int)  
>>> integer_collection[:]  
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)  
3
```

Returns count.

(TypedList) .**extend** (*tokens*)  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(  
...     item_class=int)  
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))  
>>> integer_collection[:]  
[0, 1, 2, 3]
```

Returns none.

(TypedList) .**index** (*token*)  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(  
...     tokens=('c'qf', "as'", 'b', 'dss'),  
...     item_class=pitchtools.NamedPitch)  
>>> pitch_collection.index("as'")  
1
```

Returns index.

(TypedList) .**insert** (*i*, *token*)  
Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(  
...     item_class=int)  
>>> integer_collection.extend(('1', 2, 4.3))  
>>> integer_collection[:]  
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

(TypedList) **.pop** (*i=-1*)

Aliases list.pop().

(TypedList) **.remove** (*token*)

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()

Aliases list.reverse().

(TypedList) **.sort** (*cmp=None, key=None, reverse=False*)

Aliases list.sort().

## Special methods

(TypedCollection) **.\_\_contains\_\_** (*token*)

(TypedList) **.\_\_delitem\_\_** (*i*)

Aliases list.\_\_delitem\_\_().

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedList) **.\_\_getitem\_\_** (*i*)

Aliases list.\_\_getitem\_\_().

(TypedList) **.\_\_iadd\_\_** (*expr*)

Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mp',
        target_context=stafftools.Staff
    )
])
```

```

    ),
    contexttools.DynamicMark(
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'fff',
        target_context=stafftools.Staff
    )
],
item_class=contexttools.DynamicMark
)

```

Returns collection.

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

(TypedCollection).**\_\_ne\_\_**(*expr*)

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList).**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList).**\_\_setitem\_\_**(*i*, *expr*)

Change tokens in *expr* to items and set:

```

>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch('gqs,')
],
item_class=pitchtools.NamedPitch
)

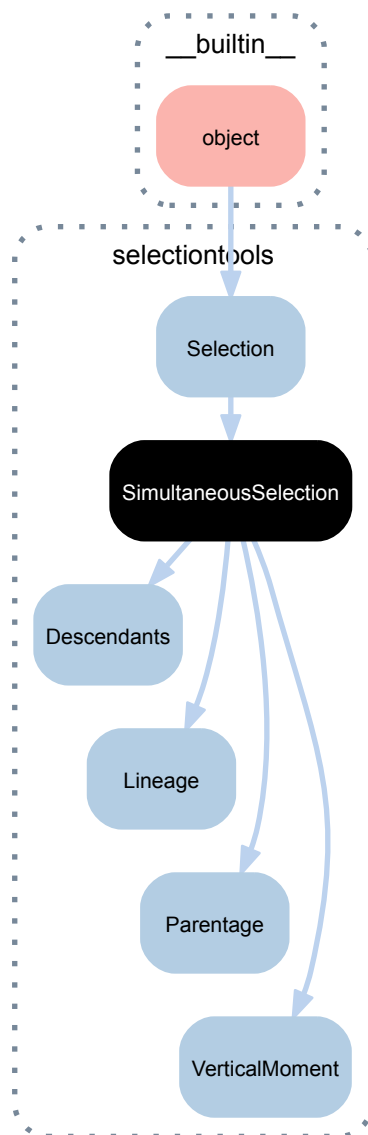
```

```

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch("f'"),
    pitchtools.NamedPitch("g'"),
    pitchtools.NamedPitch("a'"),
    pitchtools.NamedPitch("b'"),
    pitchtools.NamedPitch("c'")
],
item_class=pitchtools.NamedPitch
)

```

### 28.1.7 selectiontools.SimultaneousSelection



**class** `selectiontools.SimultaneousSelection` (*music=None*)

SliceSelection of components taken simultaneously.

Simultaneously selections implement no duration properties.

#### Bases

- `selectiontools.Selection`
- `__builtin__.object`

#### Methods

(`Selection`) **.get\_duration** (*in\_seconds=False*)

Gets duration of contiguous selection.

Returns duration.

(`Selection`) **.get\_spanners** (*spanner\_classes=None*)

Gets spanners attached to any component in selection.

Returns set.

`SimultaneousSelection.get_vertical_moment_at` (*offset*)  
Select vertical moment at *offset*.

### Special methods

`(Selection).__add__(expr)`  
Cocatenates *expr* to selection.

Returns new selection.

`(Selection).__contains__(expr)`  
True when *expr* is in selection. Otherwise false.

Returns boolean.

`(Selection).__eq__(expr)`  
True when selection and *expr* are of the same type and when music of selection equals music of *expr*.  
Otherwise false.

Returns boolean.

`(Selection).__getitem__(expr)`  
Gets item *expr* from selection.

Returns component from selection.

`(Selection).__len__()`  
Number of components in selection.

Returns nonnegative integer.

`(Selection).__ne__(expr)`  
True when selection does not equal *expr*. Otherwise false.

Returns boolean.

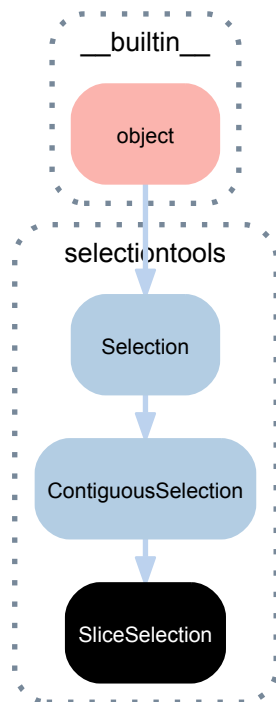
`(Selection).__radd__(expr)`  
Concatenates selection to *expr*.

Returns newly created selection.

`(Selection).__repr__()`  
Interpreter representation of selection.

Returns string.

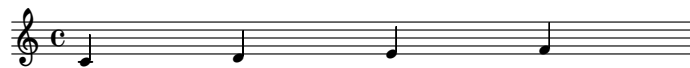
## 28.1.8 selectiontools.SliceSelection



**class** `selectiontools.SliceSelection` (*music=None*)  
 A time-contiguous selection of components all in the same parent.

### Example.

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> show(staff)
```



```
>>> staff[:2]
SliceSelection(Note("c'4"), Note("d'4"))
```

### Bases

- `selectiontools.ContiguousSelection`
- `selectiontools.Selection`
- `__builtin__.object`

### Methods

(`Selection`) **.get\_duration** (*in\_seconds=False*)  
 Gets duration of contiguous selection.

Returns duration.

(`Selection`) **.get\_spanners** (*spanner\_classes=None*)  
 Gets spanners attached to any component in selection.

Returns set.

(`ContiguousSelection`) **.get\_timespan** (*in\_seconds=False*)  
 Gets timespan of contiguous selection.

Returns timespan.

(ContiguousSelection) **.group\_by** (*predicate*)

Groups components in contiguous selection by *predicate*.

Returns list of tuples.

(ContiguousSelection) **.partition\_by\_durations** (*durations*, *cyclic=False*, *fill='exact'*,  
*in\_seconds=False*, *overhang=False*)

Partitions *components* according to *durations*.

When *fill* is 'exact' then parts must equal *durations* exactly.

When *fill* is 'less' then parts must be less than or equal to *durations*.

When *fill* is 'greater' then parts must be greater or equal to *durations*.

Reads *durations* cyclically when *cyclic* is true.

Reads component durations in seconds when *in\_seconds* is true.

Returns remaining components at end in final part when *overhang* is true.

(ContiguousSelection) **.partition\_by\_durations\_exactly** (*durations*, *cyclic=False*,  
*in\_seconds=False*, *overhang=False*)

(ContiguousSelection) **.partition\_by\_durations\_not\_greater\_than** (*durations*,  
*cyclic=False*,  
*in\_seconds=False*,  
*overhang=False*)

(ContiguousSelection) **.partition\_by\_durations\_not\_less\_than** (*durations*,  
*cyclic=False*,  
*in\_seconds=False*,  
*overhang=False*)

## Special methods

(ContiguousSelection) **.\_\_add\_\_** (*expr*)

Add *expr* to slice selection.

Returns new slice selection.

(Selection) **.\_\_contains\_\_** (*expr*)

True when *expr* is in selection. Otherwise false.

Returns boolean.

(Selection) **.\_\_eq\_\_** (*expr*)

True when selection and *expr* are of the same type and when music of selection equals music of *expr*. Otherwise false.

Returns boolean.

(Selection) **.\_\_getitem\_\_** (*expr*)

Gets item *expr* from selection.

Returns component from selection.

(Selection) **.\_\_len\_\_** ()

Number of components in selection.

Returns nonnegative integer.

(Selection) **.\_\_ne\_\_** (*expr*)

True when selection does not equal *expr*. Otherwise false.



Returns boolean.

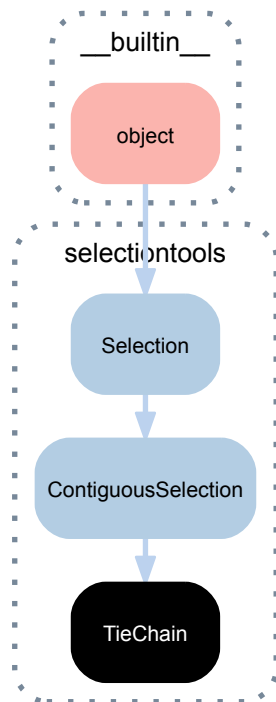
(ContiguousSelection).**\_\_radd\_\_**(*expr*)  
Add slice selection to *expr*.

Returns new slice selection.

(Selection).**\_\_repr\_\_**()  
Interpreter representation of selection.

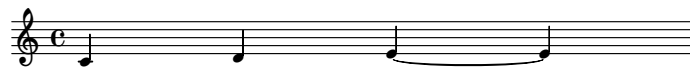
Returns string.

### 28.1.9 selectiontools.TieChain



**class** selectiontools.**TieChain**(*music=None*)  
All the notes in a tie chain.

```
>>> staff = Staff("c' d' e' ~ e'")
>>> show(staff)
```



```
>>> inspect(staff[2]).get_tie_chain()
TieChain(Note("e'4"), Note("e'4"))
```

#### Bases

- selectiontools.ContiguousSelection
- selectiontools.Selection
- \_\_builtin\_\_.object

## Read-only properties

**TieChain.all\_leaves\_are\_in\_same\_parent**  
True when all leaves in tie chain are in same parent.

Returns boolean.

**TieChain.head**  
Reference to element 0 in tie chain.

Returns component.

**TieChain.is\_pitched**  
True when tie chain head is a note or chord.

Returns boolean.

**TieChain.is\_trivial**  
True when length of tie chain is less than or equal to 1.

Returns boolean.

**TieChain.leaves**  
Leaves in tie chain.

Returns tuple.

**TieChain.leaves\_grouped\_by\_immediate\_parents**  
Leaves in tie chain grouped by immediate parents of leaves.

Returns list of lists.

**TieChain.tie\_spanner**  
Tie spanner governing tie chain.

Returns tie spanner.

**TieChain.written\_duration**  
Sum of written duration of all components in tie chain.

Returns duration.

## Methods

**(Selection).get\_duration** (*in\_seconds=False*)  
Gets duration of contiguous selection.

Returns duration.

**(Selection).get\_spanners** (*spanner\_classes=None*)  
Gets spanners attached to any component in selection.

Returns set.

**(ContiguousSelection).get\_timespan** (*in\_seconds=False*)  
Gets timespan of contiguous selection.

Returns timespan.

**(ContiguousSelection).group\_by** (*predicate*)  
Groups components in contiguous selection by *predicate*.

Returns list of tuples.

**(ContiguousSelection).partition\_by\_durations** (*durations*, *cyclic=False*, *fill='exact'*,  
*in\_seconds=False*, *overhang=False*)

Partitions *components* according to *durations*.

When *fill* is 'exact' then parts must equal *durations* exactly.

When *fill* is 'less' then parts must be less than or equal to *durations*.

When *fill* is 'greater' then parts must be greater or equal to *durations*.

Reads *durations* cyclically when *cyclic* is true.

Reads component durations in seconds when *in\_seconds* is true.

Returns remaining components at end in final part when *overhang* is true.

```
(ContiguousSelection).partition_by_durations_exactly(durations, cyclic=False,
                                                    in_seconds=False, overhang=False)
```

```
(ContiguousSelection).partition_by_durations_not_greater_than(durations,
                                                             cyclic=False,
                                                             in_seconds=False,
                                                             overhang=False)
```

```
(ContiguousSelection).partition_by_durations_not_less_than(durations,
                                                           cyclic=False,
                                                           in_seconds=False,
                                                           overhang=False)
```

**TieChain.to\_tuplet** (*proportions*, *dotted=False*, *is\_diminution=True*)

Change tie chain to tuplet.

**Example 1.** Change tie chain to diminished tuplet:

```
>>> staff = Staff(r"c'8 ~ c'16 cqs''4")
>>> crescendo = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> staff.override.dynamic_line_spanner.staff_padding = 3
>>> time_signature = contexttools.TimeSignatureMark((7, 16))
>>> time_signature.attach(staff)
TimeSignatureMark((7, 16))(Staff{3})
```

```
>>> show(staff)
```



```
>>> tie_chain = inspect(staff[0]).get_tie_chain()
>>> tie_chain.to_tuplet([2, 1, 1, 1], is_diminution=True)
FixedDurationTuplet(3/16, [c'8, c'16, c'16, c'16])
```

```
>>> show(staff)
```



**Example 2.** Change tie chain to augmented tuplet:

```
>>> staff = Staff(r"c'8 ~ c'16 cqs''4")
>>> crescendo = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> staff.override.dynamic_line_spanner.staff_padding = 3
>>> time_signature = contexttools.TimeSignatureMark((7, 16))
>>> time_signature.attach(staff)
TimeSignatureMark((7, 16))(Staff{3})
```

```
>>> show(staff)
```



```
>>> tie_chain = inspect(staff[0]).get_tie_chain()
>>> tie_chain.to_tuplet([2, 1, 1, 1], is_diminution=False)
FixedDurationTuplet(3/16, [c'16, c'32, c'32, c'32])
```

```
>>> show(staff)
```



Returns tuplet.

## Special methods

(ContiguousSelection).**\_\_add\_\_**(*expr*)

Add *expr* to slice selection.

Returns new slice selection.

(Selection).**\_\_contains\_\_**(*expr*)

True when *expr* is in selection. Otherwise false.

Returns boolean.

(Selection).**\_\_eq\_\_**(*expr*)

True when selection and *expr* are of the same type and when music of selection equals music of *expr*. Otherwise false.

Returns boolean.

(Selection).**\_\_getitem\_\_**(*expr*)

Gets item *expr* from selection.

Returns component from selection.

(Selection).**\_\_len\_\_**()

Number of components in selection.

Returns nonnegative integer.

(Selection).**\_\_ne\_\_**(*expr*)

True when selection does not equal *expr*. Otherwise false.

Returns boolean.

(ContiguousSelection).**\_\_radd\_\_**(*expr*)

Add slice selection to *expr*.

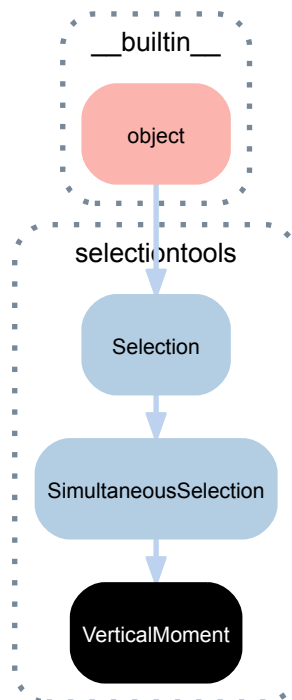
Returns new slice selection.

(Selection).**\_\_repr\_\_**()

Interpreter representation of selection.

Returns string.

### 28.1.10 selectiontools.VerticalMoment



**class** selectiontools.**VerticalMoment** (*expr=None, offset=None*)  
 Everything happening at a single moment in musical time:

```
>>> score = Score([])
>>> piano_staff = scoretools.PianoStaff([])
>>> piano_staff.append(Staff("c'4 e'4 d'4 f'4"))
>>> piano_staff.append(Staff(r"""\clef "bass" g2 f2"""))
>>> score.append(piano_staff)
```

```
>>> show(score)
```



```
>>> for x in iterationtools.iterate_vertical_moments_in_expr(score):
...     x
...
VerticalMoment(0, <<2>>)
VerticalMoment(1/4, <<2>>)
VerticalMoment(1/2, <<2>>)
VerticalMoment(3/4, <<2>>)
```

#### Bases

- selectiontools.SimultaneousSelection
- selectiontools.Selection
- \_\_builtin\_\_.object

#### Read-only properties

VerticalMoment.**attack\_count**

Positive integer number of pitch carriers starting at vertical moment.

`VerticalMoment.components`

Tuple of zero or more components happening at vertical moment.

It is always the case that `self.components = self.overlap_components + self.start_components`.

`VerticalMoment.governors`

Tuple of one or more containers in which vertical moment is evaluated.

`VerticalMoment.leaves`

Tuple of zero or more leaves at vertical moment.

`VerticalMoment.measures`

Tuplet of zero or more measures at vertical moment.

`VerticalMoment.next_vertical_moment`

Reference to next vertical moment forward in time.

`VerticalMoment.notes`

Tuple of zero or more notes at vertical moment.

`VerticalMoment.offset`

Rational-valued score offset at which vertical moment is evaluated.

`VerticalMoment.overlap_components`

Tuple of components in vertical moment starting before vertical moment, ordered by score index.

`VerticalMoment.overlap_leaves`

Tuple of leaves in vertical moment starting before vertical moment, ordered by score index.

`VerticalMoment.overlap_measures`

Tuple of measures in vertical moment starting before vertical moment, ordered by score index.

`VerticalMoment.overlap_notes`

Tuple of notes in vertical moment starting before vertical moment, ordered by score index.

`VerticalMoment.previous_vertical_moment`

Reference to prev vertical moment backward in time.

`VerticalMoment.start_components`

Tuple of components in vertical moment starting with at vertical moment, ordered by score index.

`VerticalMoment.start_leaves`

Tuple of leaves in vertical moment starting with vertical moment, ordered by score index.

`VerticalMoment.start_notes`

Tuple of notes in vertical moment starting with vertical moment, ordered by score index.

## Methods

`(Selection).get_duration(in_seconds=False)`

Gets duration of contiguous selection.

Returns duration.

`(Selection).get_spanners(spanner_classes=None)`

Gets spanners attached to any component in selection.

Returns set.

`(SimultaneousSelection).get_vertical_moment_at(offset)`

Select vertical moment at *offset*.

## Special methods

`(Selection).__add__(expr)`  
Cocatenates *expr* to selection.  
Returns new selection.

`(Selection).__contains__(expr)`  
True when *expr* is in selection. Otherwise false.  
Returns boolean.

`VerticalMoment.__eq__(expr)`

`(Selection).__getitem__(expr)`  
Gets item *expr* from selection.  
Returns component from selection.

`VerticalMoment.__hash__()`

`VerticalMoment.__len__()`

`VerticalMoment.__ne__(expr)`

`(Selection).__radd__(expr)`  
Concatenates selection to *expr*.  
Returns newly created selection.

`VerticalMoment.__repr__()`

## 28.2 Functions

### 28.2.1 selectiontools.select

`selectiontools.select(expr=None, contiguous=False)`  
Select *expr*.  
Returns selection.





# SEQUENCETOOLS

## 29.1 Functions

### 29.1.1 `sequencetools.all_are_assignable_integers`

`sequencetools.all_are_assignable_integers` (*expr*)

True when *expr* is a sequence and all elements in *expr* are notehead-assignable integers:

```
>>> sequencetools.all_are_assignable_integers([1, 2, 3, 4, 6, 7, 8, 12, 14, 15, 16])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_assignable_integers([])
True
```

False otherwise:

```
>>> sequencetools.all_are_assignable_integers('foo')
False
```

Returns boolean.

### 29.1.2 `sequencetools.all_are_equal`

`sequencetools.all_are_equal` (*expr*)

True when *expr* is a sequence and all elements in *expr* are equal:

```
>>> sequencetools.all_are_equal([99, 99, 99, 99, 99, 99])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_equal([])
True
```

False otherwise:

```
>>> sequencetools.all_are_equal(17)
False
```

Returns boolean.

### 29.1.3 `sequencetools.all_are_integer_equivalent_exprs`

`sequencetools.all_are_integer_equivalent_exprs` (*expr*)

True when *expr* is a sequence and all elements in *expr* are integer-equivalent expressions:

```
>>> sequencetools.all_are_integer_equivalent_exprs([1, '2', 3.0, Fraction(4, 1)])
True
```

Otherwise false:

```
>>> sequencetools.all_are_integer_equivalent_exprs([1, '2', 3.5, 4])
False
```

Returns boolean.

### 29.1.4 `sequencetools.all_are_integer_equivalent_numbers`

`sequencetools.all_are_integer_equivalent_numbers(expr)`

True when *expr* is a sequence and all elements in *expr* are integer-equivalent numbers:

```
>>> sequencetools.all_are_integer_equivalent_numbers([1, 2, 3.0, Fraction(4, 1)])
True
```

Otherwise false:

```
>>> sequencetools.all_are_integer_equivalent_numbers([1, 2, 3.5, 4])
False
```

Returns boolean.

### 29.1.5 `sequencetools.all_are_nonnegative_integer_equivalent_numbers`

`sequencetools.all_are_nonnegative_integer_equivalent_numbers(expr)`

True *expr* is a sequence and when all elements in *expr* are nonnegative integer-equivalent numbers. Otherwise false:

```
>>> expr = [0, 0.0, Fraction(0), 2, 2.0, Fraction(2)]
>>> sequencetools.all_are_nonnegative_integer_equivalent_numbers(expr)
True
```

Returns boolean.

### 29.1.6 `sequencetools.all_are_nonnegative_integer_powers_of_two`

`sequencetools.all_are_nonnegative_integer_powers_of_two(expr)`

True when *expr* is a sequence and all elements in *expr* are nonnegative integer powers of two:

```
>>> sequencetools.all_are_nonnegative_integer_powers_of_two([0, 1, 1, 1, 2, 4, 32, 32])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_nonnegative_integer_powers_of_two([])
True
```

False otherwise:

```
>>> sequencetools.all_are_nonnegative_integer_powers_of_two(17)
False
```

Returns boolean.

### 29.1.7 `sequencetools.all_are_nonnegative_integers`

`sequencetools.all_are_nonnegative_integers(expr)`

True when *expr* is a sequence and all elements in *expr* are nonnegative integers:

```
>>> sequencetools.all_are_nonnegative_integers([0, 1, 2, 99])
True
```

Otherwise false:

```
>>> sequencetools.all_are_nonnegative_integers([0, 1, 2, -99])
False
```

Returns boolean.

### 29.1.8 sequencetools.all\_are\_numbers

`sequencetools.all_are_numbers(expr)`

True when *expr* is a sequence and all elements in *expr* are numbers:

```
>>> sequencetools.all_are_numbers([1, 2, 3.0, Fraction(13, 8)])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_numbers([])
True
```

False otherwise:

```
>>> sequencetools.all_are_numbers(17)
False
```

Returns boolean.

### 29.1.9 sequencetools.all\_are\_pairs

`sequencetools.all_are_pairs(expr)`

True when *expr* is a sequence whose members are all sequences of length 2:

```
>>> sequencetools.all_are_pairs([(1, 2), (3, 4), (5, 6), (7, 8)])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_pairs([])
True
```

False otherwise:

```
>>> sequencetools.all_are_pairs('foo')
False
```

Returns boolean.

### 29.1.10 sequencetools.all\_are\_pairs\_of\_types

`sequencetools.all_are_pairs_of_types(expr, first_type, second_type)`

True when *expr* is a sequence whose members are all sequences of length 2, and where the first member of each pair is an instance of *first\_type* and where the second member of each pair is an instance of *second\_type*:

```
>>> sequencetools.all_are_pairs_of_types([(1., 'a'), (2.1, 'b'), (3.45, 'c')], float, str)
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_pairs_of_types([], float, str)
True
```

False otherwise:

```
>>> sequencetools.all_are_pairs_of_types('foo', float, str)
False
```

Returns boolean.

### 29.1.11 `sequencetools.all_are_positive_integer_equivalent_numbers`

`sequencetools.all_are_positive_integer_equivalent_numbers(expr)`

True when *expr* is a sequence and all elements in *expr* are positive integer-equivalent numbers. Otherwise false:

```
>>> sequencetools.all_are_positive_integer_equivalent_numbers([Fraction(4, 2), 2.0, 2])
True
```

Returns boolean.

### 29.1.12 `sequencetools.all_are_positive_integers`

`sequencetools.all_are_positive_integers(expr)`

True when *expr* is a sequence and all elements in *expr* are positive integers:

```
>>> sequencetools.all_are_positive_integers([1, 2, 3, 99])
True
```

Otherwise false:

```
>>> sequencetools.all_are_positive_integers(17)
False
```

Returns boolean.

### 29.1.13 `sequencetools.all_are_unequal`

`sequencetools.all_are_unequal(expr)`

True when *expr* is a sequence all elements in *expr* are unequal:

```
>>> sequencetools.all_are_unequal([1, 2, 3, 4, 9])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_unequal([])
True
```

False otherwise:

```
>>> sequencetools.all_are_unequal(17)
False
```

Returns boolean.

### 29.1.14 `sequencetools.count_length_two_runs_in_sequence`

`sequencetools.count_length_two_runs_in_sequence(sequence)`

Count length-2 runs in *sequence*:

```
>>> sequencetools.count_length_two_runs_in_sequence([0, 0, 1, 1, 1, 2, 3, 4, 5])
3
```

Returns nonnegative integer.

### 29.1.15 sequencetools.divide\_sequence\_elements\_by\_greatest\_common\_divisor

`sequencetools.divide_sequence_elements_by_greatest_common_divisor(sequence)`  
Divide *sequence* elements by greatest common divisor:

```
>>> sequencetools.divide_sequence_elements_by_greatest_common_divisor([2, 2, -8, -16])
[1, 1, -4, -8]
```

Allow negative *sequence* elements.

Raise type error on noninteger *sequence* elements.

Raise not implemented error when 0 in *sequence*.

Returns new *sequence* object.

### 29.1.16 sequencetools.flatten\_sequence

`sequencetools.flatten_sequence(sequence, classes=None, depth=-1)`  
Flattens *sequence*.

Flatten *sequence* completely:

```
>>> sequence = [1, [2, 3, [4]], 5, [6, 7, [8]]]
>>> sequencetools.flatten_sequence(sequence)
[1, 2, 3, 4, 5, 6, 7, 8]
```

Flatten *sequence* to depth 1:

```
>>> sequence = [1, [2, 3, [4]], 5, [6, 7, [8]]]
>>> sequencetools.flatten_sequence(sequence, depth=1)
[1, 2, 3, [4], 5, 6, 7, [8]]
```

Flatten *sequence* to depth 2:

```
>>> sequence = [1, [2, 3, [4]], 5, [6, 7, [8]]]
>>> sequencetools.flatten_sequence(sequence, depth=2)
[1, 2, 3, 4, 5, 6, 7, 8]
```

Leaves *sequence* unchanged.

Returns new object of *sequence* type.

### 29.1.17 sequencetools.flatten\_sequence\_at\_indices

`sequencetools.flatten_sequence_at_indices(sequence, indices, classes=None, depth=-1)`  
Flatten *sequence* at *indices*:

```
>>> sequencetools.flatten_sequence_at_indices([0, 1, [2, 3, 4], [5, 6, 7]], [3])
[0, 1, [2, 3, 4], 5, 6, 7]
```

Flatten *sequence* at negative *indices*:

```
>>> sequencetools.flatten_sequence_at_indices([0, 1, [2, 3, 4], [5, 6, 7]], [-1])
[0, 1, [2, 3, 4], 5, 6, 7]
```

Leave *sequence* unchanged.

Returns newly constructed *sequence* object.

### 29.1.18 sequencetools.get\_indices\_of\_sequence\_elements\_equal\_to\_true

`sequencetools.get_indices_of_sequence_elements_equal_to_true(sequence)`  
Get indices of *sequence* elements equal to true:

```
>>> sequence = [0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1]
```

```
>>> sequencetools.get_indices_of_sequence_elements_equal_to_true(sequence)
(3, 4, 5, 9, 10, 11, 12)
```

Returns newly constructed tuple of zero or more nonnegative integers.

### 29.1.19 sequencetools.get\_sequence\_degree\_of\_rotational\_symmetry

`sequencetools.get_sequence_degree_of_rotational_symmetry(sequence)`  
Change *sequence* to degree of rotational symmetry:

```
>>> sequencetools.get_sequence_degree_of_rotational_symmetry([1, 2, 3, 4, 5, 6])
1
```

```
>>> sequencetools.get_sequence_degree_of_rotational_symmetry([1, 2, 3, 1, 2, 3])
2
```

```
>>> sequencetools.get_sequence_degree_of_rotational_symmetry([1, 2, 1, 2, 1, 2])
3
```

```
>>> sequencetools.get_sequence_degree_of_rotational_symmetry([1, 1, 1, 1, 1, 1])
6
```

Returns positive integer.

### 29.1.20 sequencetools.get\_sequence\_element\_at\_cyclic\_index

`sequencetools.get_sequence_element_at_cyclic_index(sequence, index)`  
Get *sequence* element at nonnegative cyclic *index*:

```
>>> for index in range(10):
...     print '%s\t%s' % (index, sequencetools.get_sequence_element_at_cyclic_index(
...         'string', index))
...
0 s
1 t
2 r
3 i
4 n
5 g
6 s
7 t
8 r
9 i
```

Get *sequence* element at negative cyclic *index*:

```
>>> for index in range(1, 11):
...     print '%s\t%s' % (-index, sequencetools.get_sequence_element_at_cyclic_index(
...         'string', -index))
...
-1 g
-2 n
-3 i
-4 r
-5 t
-6 s
-7 g
-8 n
-9 i
-10 r
```

Returns reference to *sequence* element.

### 29.1.21 sequencetools.get\_sequence\_elements\_at\_indices

`sequencetools.get_sequence_elements_at_indices(sequence, indices)`

Get *sequence* elements at *indices*:

```
>>> sequencetools.get_sequence_elements_at_indices('string of text', (2, 3, 10, 12))
('r', 'i', 't', 'x')
```

Returns newly constructed tuple of references to *sequence* elements.

### 29.1.22 sequencetools.get\_sequence\_elements\_frequency\_distribution

`sequencetools.get_sequence_elements_frequency_distribution(sequence)`

Get *sequence* elements frequency distribution:

```
>>> sequence = [1, 3, 3, 3, 2, 1, 1, 2, 3, 3, 1, 2]
```

```
>>> sequencetools.get_sequence_elements_frequency_distribution(sequence)
[(1, 4), (2, 3), (3, 5)]
```

Returns list of element / count pairs.

### 29.1.23 sequencetools.get\_sequence\_period\_of\_rotation

`sequencetools.get_sequence_period_of_rotation(sequence, n)`

Change *sequence* to period of rotation:

```
>>> sequencetools.get_sequence_period_of_rotation([1, 2, 3, 1, 2, 3], 1)
3
```

```
>>> sequencetools.get_sequence_period_of_rotation([1, 2, 3, 1, 2, 3], 2)
3
```

```
>>> sequencetools.get_sequence_period_of_rotation([1, 2, 3, 1, 2, 3], 3)
1
```

Returns positive integer.

### 29.1.24 sequencetools.increase\_sequence\_elements\_at\_indices\_by\_addenda

`sequencetools.increase_sequence_elements_at_indices_by_addenda(sequence, addenda, indices)`

Increase *sequence* by *addenda* at *indices*:

```
>>> sequence = [1, 1, 2, 3, 5, 5, 1, 2, 5, 5, 6]
```

```
>>> sequencetools.increase_sequence_elements_at_indices_by_addenda(
...     sequence, [0.5, 0.5], [0, 4, 8])
[1.5, 1.5, 2, 3, 5.5, 5.5, 1, 2, 5.5, 5.5, 6]
```

Returns list.

### 29.1.25 sequencetools.increase\_sequence\_elements\_cyclically\_by\_addenda

`sequencetools.increase_sequence_elements_cyclically_by_addenda` (*sequence*,  
*addenda*,  
*shield=True*)

Increase *sequence* cyclically by *addenda*:

```
>>> sequencetools.increase_sequence_elements_cyclically_by_addenda(  
...     range(10), [10, -10], shield=False)  
[10, -9, 12, -7, 14, -5, 16, -3, 18, -1]
```

Increase *sequence* cyclically by *addenda* and map nonpositive values to 1:

```
>>> sequencetools.increase_sequence_elements_cyclically_by_addenda(  
...     range(10), [10, -10], shield=True)  
[10, 1, 12, 1, 14, 1, 16, 1, 18, 1]
```

Returns list.

### 29.1.26 sequencetools.interlace\_sequences

`sequencetools.interlace_sequences` (*\*sequences*)

Interlace *sequences*:

```
>>> k = range(100, 103)  
>>> l = range(200, 201)  
>>> m = range(300, 303)  
>>> n = range(400, 408)  
>>> sequencetools.interlace_sequences(k, l, m, n)  
[100, 200, 300, 400, 101, 301, 401, 102, 302, 402, 403, 404, 405, 406, 407]
```

Returns list.

### 29.1.27 sequencetools.is\_fraction\_equivalent\_pair

`sequencetools.is_fraction_equivalent_pair` (*expr*)

True when *expr* is an integer-equivalent pair of numbers excluding 0 as the second term:

```
>>> sequencetools.is_fraction_equivalent_pair((2, 3))  
True
```

Otherwise false:

```
>>> sequencetools.is_fraction_equivalent_pair((2, 0))  
False
```

Returns boolean.

### 29.1.28 sequencetools.is\_integer\_equivalent\_n\_tuple

`sequencetools.is_integer_equivalent_n_tuple` (*expr*, *n*)

True when *expr* is a tuple of *n* integer-equivalent expressions:

```
>>> sequencetools.is_integer_equivalent_n_tuple((2.0, '3', Fraction(4, 1)), 3)  
True
```

Otherwise false:

```
>>> sequencetools.is_integer_equivalent_n_tuple((2.5, '3', Fraction(4, 1)), 3)  
False
```

Returns boolean.



### 29.1.29 sequencetools.is\_integer\_equivalent\_pair

`sequencetools.is_integer_equivalent_pair` (*expr*)

True when *expr* is a pair of integer-equivalent expressions:

```
>>> sequencetools.is_integer_equivalent_pair((2.0, '3'))
True
```

Otherwise false:

```
>>> sequencetools.is_integer_equivalent_pair((2.5, '3'))
False
```

Returns boolean.

### 29.1.30 sequencetools.is\_integer\_equivalent\_singleton

`sequencetools.is_integer_equivalent_singleton` (*expr*)

True when *expr* is a singleton of integer-equivalent expressions:

```
>>> sequencetools.is_integer_equivalent_singleton((2.0,))
True
```

Otherwise false:

```
>>> sequencetools.is_integer_equivalent_singleton((2.5,))
False
```

Returns boolean.

### 29.1.31 sequencetools.is\_integer\_n\_tuple

`sequencetools.is_integer_n_tuple` (*expr*, *n*)

True when *expr* is an integer tuple of length *n*:

```
>>> sequencetools.is_integer_n_tuple((19, 20, 21), 3)
True
```

Otherwise false:

```
>>> sequencetools.is_integer_n_tuple((19, 20, 'text'), 3)
False
```

Returns boolean.

### 29.1.32 sequencetools.is\_integer\_pair

`sequencetools.is_integer_pair` (*expr*)

True when *expr* is an integer tuple of length 2:

```
>>> sequencetools.is_integer_pair((19, 20))
True
```

Otherwise false:

```
>>> sequencetools.is_integer_pair(('some', 'text'))
False
```

Returns boolean.

### 29.1.33 sequencetools.is\_integer\_singleton

`sequencetools.is_integer_singleton(expr)`

True when *expr* is an integer tuple of length 1:

```
>>> sequencetools.is_integer_singleton((19,))
True
```

Otherwise false:

```
>>> sequencetools.is_integer_singleton('text',)
False
```

Returns boolean.

### 29.1.34 sequencetools.is\_monotonically\_decreasing\_sequence

`sequencetools.is_monotonically_decreasing_sequence(expr)`

True when *expr* is a sequence and the elements in *expr* decrease monotonically:

```
>>> expr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
True
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 2, 1, 0]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
True
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
True
```

False when *expr* is a sequence and the elements in *expr* do not decrease monotonically:

```
>>> expr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
False
```

```
>>> expr = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
False
```

True when *expr* is a sequence and *expr* is empty:

```
>>> expr = []
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
True
```

False when *expr* is not a sequence:

```
>>> sequencetools.is_monotonically_decreasing_sequence(17)
False
```

Returns boolean.

### 29.1.35 sequencetools.is\_monotonically\_increasing\_sequence

`sequencetools.is_monotonically_increasing_sequence(expr)`

True when *expr* is a sequence and the elements in *expr* increase monotonically:

```
>>> expr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
True
```

```
>>> expr = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
True
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
True
```

False when *expr* is a sequence and the elements in *expr* do not increase monotonically:

```
>>> expr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 2, 1, 0]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
False
```

True when *expr* is a sequence and *expr* is empty:

```
>>> expr = []
>>> sequencetools.is_monotonically_increasing_sequence(expr)
True
```

False when *expr* is not a sequence:

```
>>> sequencetools.is_monotonically_increasing_sequence(17)
False
```

Returns boolean.

### 29.1.36 sequencetools.is\_n\_tuple

`sequencetools.is_n_tuple(expr, n)`

True when *expr* is a tuple of length *n*:

```
>>> sequencetools.is_n_tuple((19, 20, 21), 3)
True
```

Otherwise false:

```
>>> sequencetools.is_n_tuple((19, 20, 21), 4)
False
```

Returns boolean.

### 29.1.37 sequencetools.is\_null\_tuple

`sequencetools.is_null_tuple(expr)`

True when *expr* is a tuple of length 0:

```
>>> sequencetools.is_null_tuple(())
True
```

Otherwise false:

```
>>> sequencetools.is_null_tuple((19, 20, 21))
False
```

Returns boolean.

### 29.1.38 sequencetools.is\_pair

`sequencetools.is_pair(expr)`

True when *expr* is a tuple of length 2:

```
>>> sequencetools.is_pair((19, 20))
True
```

Otherwise false:

```
>>> sequencetools.is_pair((19, 20, 21))
False
```

Returns boolean.

### 29.1.39 sequencetools.is\_permutation

`sequencetools.is_permutation(expr, length=None)`

True when *expr* is a permutation:

```
>>> sequencetools.is_permutation([4, 5, 0, 3, 2, 1])
True
```

Otherwise false:

```
>>> sequencetools.is_permutation([1, 1, 5, 3, 2, 1])
False
```

True when *expr* is a permutation of first *length* nonnegative integers:

```
>>> sequencetools.is_permutation([4, 5, 0, 3, 2, 1], length=6)
True
```

Otherwise false:

```
>>> sequencetools.is_permutation([4, 0, 3, 2, 1], length=6)
False
```

Returns boolean.

### 29.1.40 sequencetools.is\_repetition\_free\_sequence

`sequencetools.is_repetition_free_sequence(expr)`

True when *expr* is a sequence and *expr* is repetition free:

```
>>> sequencetools.is_repetition_free_sequence([0, 1, 2, 6, 7, 8])
True
```

False when *expr* is a sequence and *expr* is not repetition free:

```
>>> sequencetools.is_repetition_free_sequence([0, 1, 2, 2, 7, 8])
False
```

True when *expr* is an empty sequence:

```
>>> sequencetools.is_repetition_free_sequence([])
True
```

False *expr* is not a sequence:

```
>>> sequencetools.is_repetition_free_sequence(17)
False
```

Returns boolean.

### 29.1.41 sequencetools.is\_restricted\_growth\_function

`sequencetools.is_restricted_growth_function(expr)`

True when *expr* is a sequence and *expr* meets the criteria for a restricted growth function:

```
>>> sequencetools.is_restricted_growth_function([1, 1, 1, 1])
True
```

```
>>> sequencetools.is_restricted_growth_function([1, 1, 1, 2])
True
```

```
>>> sequencetools.is_restricted_growth_function([1, 1, 2, 1])
True
```

```
>>> sequencetools.is_restricted_growth_function([1, 1, 2, 2])
True
```

Otherwise false:

```
>>> sequencetools.is_restricted_growth_function([1, 1, 1, 3])
False
```

```
>>> sequencetools.is_restricted_growth_function(17)
False
```

A restricted growth function is a sequence `l` such that `l[0] == 1` and such that `l[i] <= max(l[:i]) + 1` for `1 <= i <= len(l)`.

Returns boolean.

### 29.1.42 sequencetools.is\_singleton

`sequencetools.is_singleton(expr)`  
True when *expr* is a tuple of length 1:

```
>>> sequencetools.is_singleton((19,))
True
```

Otherwise false:

```
>>> sequencetools.is_singleton((19, 20, 21))
False
```

Returns boolean.

### 29.1.43 sequencetools.is\_strictly\_decreasing\_sequence

`sequencetools.is_strictly_decreasing_sequence(expr)`  
True when *expr* is a sequence and the elements in *expr* decrease strictly:

```
>>> expr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
True
```

False when *expr* is a sequence and the elements in *expr* do not decrease strictly:

```
>>> expr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
False
```

```
>>> expr = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 2, 1, 0]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
False
```

True when *expr* is an empty sequence:

```
>>> sequencetools.is_strictly_decreasing_sequence([])
True
```

False *expr* is not a sequence:

```
>>> sequencetools.is_strictly_decreasing_sequence(17)
False
```

Returns boolean.

### 29.1.44 sequencetools.is\_strictly\_increasing\_sequence

`sequencetools.is_strictly_increasing_sequence(expr)`  
True when *expr* is a sequence and the elements in *expr* increase strictly:

```
>>> expr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sequencetools.is_strictly_increasing_sequence(expr)
True
```

False when *expr* is a sequence and the elements in *expr* do not increase strictly:

```
>>> expr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sequencetools.is_strictly_increasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 2, 1, 0]
>>> sequencetools.is_strictly_increasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_strictly_increasing_sequence(expr)
False
```

```
>>> expr = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_strictly_increasing_sequence(expr)
False
```

True when *expr* is an empty sequence:

```
>>> sequencetools.is_strictly_increasing_sequence([])
True
```

False when *expr* is not a sequence:

```
>>> sequencetools.is_strictly_increasing_sequence(17)
False
```

Returns boolean.

### 29.1.45 sequencetools.iterate\_sequence\_cyclically

`sequencetools.iterate_sequence_cyclically(sequence, step=1, start=0, length='inf')`  
Iterate *sequence* cyclically according to *step*, *start* and *length*:

```
>>> sequence = [1, 2, 3, 4, 5, 6, 7]
```

```
>>> list(sequencetools.iterate_sequence_cyclically(sequence, length=20))
[1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6]
```

```
>>> list(sequencetools.iterate_sequence_cyclically(sequence, 2, length=20))
[1, 3, 5, 7, 2, 4, 6, 1, 3, 5, 7, 2, 4, 6, 1, 3, 5, 7, 2, 4]
```

```
>>> list(sequencetools.iterate_sequence_cyclically(sequence, 2, 3, length=20))
[4, 6, 1, 3, 5, 7, 2, 4, 6, 1, 3, 5, 7, 2, 4, 6, 1, 3, 5, 7]
```

```
>>> list(sequencetools.iterate_sequence_cyclically(sequence, -2, 5, length=20))
[6, 4, 2, 7, 5, 3, 1, 6, 4, 2, 7, 5, 3, 1, 6, 4, 2, 7, 5, 3]
```

Allow generator input:

```
>>> list(sequencetools.iterate_sequence_cyclically(xrange(1, 8), -2, 5, length=20))
[6, 4, 2, 7, 5, 3, 1, 6, 4, 2, 7, 5, 3, 1, 6, 4, 2, 7, 5, 3]
```

Set *step* to jump size and direction across sequence.

Set *start* to the index of *sequence* where the function begins iterating.

Set *length* to number of elements to return. Set to 'inf' to return infinitely.

Returns generator.

### 29.1.46 sequencetools.iterate\_sequence\_cyclically\_from\_start\_to\_stop

`sequencetools.iterate_sequence_cyclically_from_start_to_stop(sequence, start, stop)`  
Iterate *sequence* cyclically from *start* to *stop*:

```
>>> list(sequencetools.iterate_sequence_cyclically_from_start_to_stop(range(20), 18, 10))
[18, 19, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Returns generator of references to *sequence* elements.

### 29.1.47 sequencetools.iterate\_sequence\_forward\_and\_backward\_nonoverlapping

`sequencetools.iterate_sequence_forward_and_backward_nonoverlapping(sequence)`  
Iterate *sequence* first forward and then backward, with first and last elements repeated:

```
>>> list(sequencetools.iterate_sequence_forward_and_backward_nonoverlapping(
...     [1, 2, 3, 4, 5]))
[1, 2, 3, 4, 5, 5, 4, 3, 2, 1]
```

Returns generator.

### 29.1.48 sequencetools.iterate\_sequence\_forward\_and\_backward\_overlapping

`sequencetools.iterate_sequence_forward_and_backward_overlapping(sequence)`  
Iterate *sequence* first forward and then backward, with first and last elements appearing only once:

```
>>> list(sequencetools.iterate_sequence_forward_and_backward_overlapping([1, 2, 3, 4, 5]))
[1, 2, 3, 4, 5, 4, 3, 2]
```

Returns generator.

### 29.1.49 sequencetools.iterate\_sequence\_nwise\_cyclic

`sequencetools.iterate_sequence_nwise_cyclic(sequence, n)`  
Iterate elements in *sequence* cyclically *n* at a time:

```
>>> g = sequencetools.iterate_sequence_nwise_cyclic(range(6), 3)
>>> for n in range(10):
...     print g.next()
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(4, 5, 0)
(5, 0, 1)
```

```
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
```

Returns generator.

### 29.1.50 `sequencetools.iterate_sequence_nwise_strict`

`sequencetools.iterate_sequence_nwise_strict(sequence, n)`

Iterate elements in *sequence* *n* at a time:

```
>>> for x in sequencetools.iterate_sequence_nwise_strict(range(10), 4):
...     x
...
(0, 1, 2, 3)
(1, 2, 3, 4)
(2, 3, 4, 5)
(3, 4, 5, 6)
(4, 5, 6, 7)
(5, 6, 7, 8)
(6, 7, 8, 9)
```

Returns generator.

### 29.1.51 `sequencetools.iterate_sequence_nwise_wrapped`

`sequencetools.iterate_sequence_nwise_wrapped(sequence, n)`

Iterate elements in *sequence* *n* at a time wrapped to beginning:

```
>>> list(sequencetools.iterate_sequence_nwise_wrapped(range(6), 3))
[(0, 1, 2), (1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 0), (5, 0, 1)]
```

Returns generator.

### 29.1.52 `sequencetools.iterate_sequence_pairwise_cyclic`

`sequencetools.iterate_sequence_pairwise_cyclic(sequence)`

Iterate *sequence* pairwise cyclic:

```
>>> generator = sequencetools.iterate_sequence_pairwise_cyclic(range(6))

>>> generator.next()
(0, 1)
>>> generator.next()
(1, 2)
>>> generator.next()
(2, 3)
>>> generator.next()
(3, 4)
>>> generator.next()
(4, 5)
>>> generator.next()
(5, 0)
>>> generator.next()
(0, 1)
>>> generator.next()
(1, 2)
```

Returns pair generator.



### 29.1.53 sequencetools.iterate\_sequence\_pairwise\_strict

`sequencetools.iterate_sequence_pairwise_strict(sequence)`

Iterate *sequence* pairwise strict:

```
>>> list(sequencetools.iterate_sequence_pairwise_strict(range(6)))
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)]
```

Returns pair generator.

### 29.1.54 sequencetools.iterate\_sequence\_pairwise\_wrapped

`sequencetools.iterate_sequence_pairwise_wrapped(sequence)`

Iterate *sequence* pairwise wrapped:

```
>>> list(sequencetools.iterate_sequence_pairwise_wrapped(range(6)))
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 0)]
```

Returns pair generator.

### 29.1.55 sequencetools.join\_subsequences

`sequencetools.join_subsequences(sequence)`

Join subsequences in *sequence*:

```
>>> sequencetools.join_subsequences([(1, 2, 3), (), (4, 5), (), (6,)])
(1, 2, 3, 4, 5, 6)
```

Returns newly constructed object of subsequence type.

### 29.1.56 sequencetools.join\_subsequences\_by\_sign\_of\_subsequence\_elements

`sequencetools.join_subsequences_by_sign_of_subsequence_elements(sequence)`

Join subsequences in *sequence* by sign:

```
>>> sequence = [[1, 2], [3, 4], [-5, -6, -7], [-8, -9, -10], [11, 12]]
>>> sequencetools.join_subsequences_by_sign_of_subsequence_elements(sequence)
[[1, 2, 3, 4], [-5, -6, -7, -8, -9, -10], [11, 12]]
```

```
>>> sequence = [[1, 2], [], [], [3, 4, 5], [6, 7]]
>>> sequencetools.join_subsequences_by_sign_of_subsequence_elements(sequence)
[[1, 2], [], [3, 4, 5, 6, 7]]
```

Returns new list.

### 29.1.57 sequencetools.map\_sequence\_elements\_to\_canonic\_tuples

`sequencetools.map_sequence_elements_to_canonic_tuples(sequence, decrease_parts_monotonically=True)`

Partition *sequence* elements into canonic parts that decrease monotonically:

```
>>> sequencetools.map_sequence_elements_to_canonic_tuples(
...     range(10))
[(0,), (1,), (2,), (3,), (4,), (4, 1), (6,), (7,), (8,), (8, 1)]
```

Partition *sequence* elements into canonic parts that increase monotonically:

```
>>> sequencetools.map_sequence_elements_to_canonic_tuples(
...     range(10), decrease_parts_monotonically=False)
[(0,), (1,), (2,), (3,), (4,), (1, 4), (6,), (7,), (8,), (1, 8)]
```

Raise type error when *sequence* is not a list.

Raise value error on noninteger elements in *sequence*.

Returns list of tuples.

### 29.1.58 `sequencetools.map_sequence_elements_to_numbered_sublists`

`sequencetools.map_sequence_elements_to_numbered_sublists(sequence)`

Map *sequence* elements to numbered sublists:

```
>>> sequencetools.map_sequence_elements_to_numbered_sublists([1, 2, -3, -4, 5])
[[1], [2, 3], [-4, -5, -6], [-7, -8, -9, -10], [11, 12, 13, 14, 15]]
```

```
>>> sequencetools.map_sequence_elements_to_numbered_sublists([1, 0, -3, -4, 5])
[[1], [], [-2, -3, -4], [-5, -6, -7, -8], [9, 10, 11, 12, 13]]
```

Note that numbering starts at 1.

Returns newly constructed list of lists.

### 29.1.59 `sequencetools.merge_duration_sequences`

`sequencetools.merge_duration_sequences(*sequences)`

Merge duration *sequences*:

```
>>> sequencetools.merge_duration_sequences([10, 10, 10], [7])
[7, 3, 10, 10]
```

Merge more duration sequences:

```
>>> sequencetools.merge_duration_sequences([10, 10, 10], [10, 10])
[10, 10, 10]
```

The idea is that each sequence element represents a duration.

Returns list.

### 29.1.60 `sequencetools.negate_absolute_value_of_sequence_elements_at_indices`

`sequencetools.negate_absolute_value_of_sequence_elements_at_indices(sequence, indices)`

Negate the absolute value of *sequence* elements at *indices*:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
```

```
>>> sequencetools.negate_sequence_elements_at_indices(sequence, [0, 1, 2])
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Returns newly constructed list.

### 29.1.61 `sequencetools.negate_absolute_value_of_sequence_elements_cyclically`

`sequencetools.negate_absolute_value_of_sequence_elements_cyclically(sequence, indices, period)`

Negate the absolute value of *sequence* elements at *indices* cyclically according to *period*:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
```

```
>>> sequencetools.negate_absolute_value_of_sequence_elements_cyclically(
...     sequence, [0, 1, 2], 5)
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Returns newly constructed list.

### 29.1.62 sequencetools.negate\_sequence\_elements\_at\_indices

`sequencetools.negate_sequence_elements_at_indices` (*sequence*, *indices*)

Negate *sequence* elements at *indices*:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
```

```
>>> sequencetools.negate_sequence_elements_at_indices(sequence, [0, 1, 2])
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Returns newly constructed list.

### 29.1.63 sequencetools.negate\_sequence\_elements\_cyclically

`sequencetools.negate_sequence_elements_cyclically` (*sequence*, *indices*, *period*)

Negate *sequence* elements at *indices* cyclically according to *period*:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
```

```
>>> sequencetools.negate_sequence_elements_cyclically(sequence, [0, 1, 2], 5)
[-1, -2, -3, 4, 5, 6, 7, 8, -9, -10]
```

Returns newly constructed list.

### 29.1.64 sequencetools.overwrite\_sequence\_elements\_at\_indices

`sequencetools.overwrite_sequence_elements_at_indices` (*sequence*, *pairs*)

Overwrite *sequence* elements at indices according to *pairs*:

```
>>> sequencetools.overwrite_sequence_elements_at_indices(range(10), [(0, 3), (5, 3)])
[0, 0, 0, 3, 4, 5, 5, 5, 8, 9]
```

Set *pairs* to a list of (anchor\_index, length) pairs.

Returns new list.

### 29.1.65 sequencetools.pair\_duration\_sequence\_elements\_with\_input\_pair\_values

`sequencetools.pair_duration_sequence_elements_with_input_pair_values` (*duration\_sequence*,  
*input\_pairs*)

Pair *duration\_sequence* elements with the values of *input\_pairs*:

```
>>> duration_sequence = [10, 10, 10, 10]
>>> input_pairs = [('red', 1), ('orange', 18), ('yellow', 200)]
```

```
>>> sequencetools.pair_duration_sequence_elements_with_input_pair_values(
...     duration_sequence, input_pairs)
[(10, 'red'), (10, 'orange'), (10, 'yellow'), (10, 'yellow')]
```

Returns a list of (*element*, *value*) output pairs.

The *input\_pairs* argument must be a list of (*value*, *duration*) pairs.

The basic idea behind the function is model which input pair value is in effect at the start of each element in *duration\_sequence*.

### 29.1.66 sequencetools.partition\_sequence\_by\_backgrounded\_weights

`sequencetools.partition_sequence_by_backgrounded_weights` (*sequence*, *weights*)

Partition *sequence* by backgrounded *weights*:

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(  
...     [-5, -15, -10], [20, 10])  
[[-5, -15], [-10]]
```

Further examples:

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(  
...     [-5, -15, -10], [5, 5, 5, 5, 5, 5])  
[[-5], [-15], [], [], [-10], []]
```

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(  
...     [-5, -15, -10], [1, 29])  
[[-5], [-15, -10]]
```

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(  
...     [-5, -15, -10], [2, 28])  
[[-5], [-15, -10]]
```

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(  
...     [-5, -15, -10], [1, 1, 1, 1, 1, 25])  
[[-5], [], [], [], [], [-15, -10]]
```

The term *backgrounded* is a short-hand concocted specifically for this function; rely on the formal definition to understand the function actually does.

Input constraint: the weight of *sequence* must equal the weight of *weights* exactly.

The signs of the elements in *sequence* are ignored.

Formal definition: partition *sequence* into *parts* such that (1.) the length of *parts* equals the length of *weights*; (2.) the elements in *sequence* appear in order in *parts*; and (3.) some final condition that is difficult to formalize.

Notionally what's going on here is that the elements of *weights* are acting as a list of successive time intervals into which the elements of *sequence* are being fit in accordance with the start offset of each *sequence* element.

The function models the grouping together of successive timespans according to which of an underlying sequence of time intervals it is in which each time span begins.

Note that, for any input to this function, the flattened output of this function always equals *sequence* exactly.

Note too that while *partition* is being used here in the sense of the other partitioning functions in the API, the distinguishing feature is this function is its ability to produce empty lists as output.

Returns list of *sequence* objects.

### 29.1.67 sequencetools.partition\_sequence\_by\_counts

`sequencetools.partition_sequence_by_counts` (*sequence*, *counts*, *cyclic=False*, *overhang=False*, *copy\_elements=False*)

Partition sequence by counts.

**Example 1a.** Partition sequence once by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts (
...     range(10),
...     [3],
...     cyclic=False,
...     overhang=False,
... )
[[0, 1, 2]]
```

**Example 1b.** Partition sequence once by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts (
...     range(16),
...     [4, 3],
...     cyclic=False,
...     overhang=False,
... )
[[0, 1, 2, 3], [4, 5, 6]]
```

**Example 2a.** Partition sequence cyclically by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts (
...     range(10),
...     [3],
...     cyclic=True,
...     overhang=False,
... )
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

**Example 2b.** Partition sequence cyclically by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts (
...     range(16),
...     [4, 3],
...     cyclic=True,
...     overhang=False,
... )
[[0, 1, 2, 3], [4, 5, 6], [7, 8, 9, 10], [11, 12, 13]]
```

**Example 3a.** Partition sequence once by counts with overhang:

```
>>> sequencetools.partition_sequence_by_counts (
...     range(10),
...     [3],
...     cyclic=False,
...     overhang=True,
... )
[[0, 1, 2], [3, 4, 5, 6, 7, 8, 9]]
```

**Example 3b.** Partition sequence once by counts with overhang:

```
>>> sequencetools.partition_sequence_by_counts (
...     range(16),
...     [4, 3],
...     cyclic=False,
...     overhang=True,
... )
[[0, 1, 2, 3], [4, 5, 6], [7, 8, 9, 10, 11, 12, 13, 14, 15]]
```

**Example 4a.** Partition sequence cyclically by counts with overhang:

```
>>> sequencetools.partition_sequence_by_counts (
...     range(10),
...     [3],
...     cyclic=True,
...     overhang=True,
... )
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

**Example 4b.** Partition sequence cyclically by counts with overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(16),
...     [4, 3],
...     cyclic=True,
...     overhang=True,
... )
[[0, 1, 2, 3], [4, 5, 6], [7, 8, 9, 10], [11, 12, 13], [14, 15]]
```

Returns list of sequence objects.

### 29.1.68 sequencetools.partition\_sequence\_by\_ratio\_of\_lengths

`sequencetools.partition_sequence_by_ratio_of_lengths(sequence, lengths)`

Partition *sequence* by ratio of *lengths*:

```
>>> sequence = tuple(range(10))
```

```
>>> sequencetools.partition_sequence_by_ratio_of_lengths(
...     sequence,
...     [1, 1, 2],
... )
[(0, 1, 2), (3, 4), (5, 6, 7, 8, 9)]
```

Use rounding magic to avoid fractional part lengths.

Returns list of *sequence* objects.

### 29.1.69 sequencetools.partition\_sequence\_by\_ratio\_of\_weights

`sequencetools.partition_sequence_by_ratio_of_weights(sequence, weights)`

Partition *sequence* by ratio of *weights*:

```
>>> sequencetools.partition_sequence_by_ratio_of_weights(
...     [1] * 10, [1, 1, 1])
[[1, 1, 1], [1, 1, 1, 1], [1, 1, 1]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights(
...     [1] * 10, [1, 1, 1, 1])
[[1, 1, 1], [1, 1], [1, 1, 1], [1, 1]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights(
...     [1] * 10, [2, 2, 3])
[[1, 1, 1], [1, 1, 1], [1, 1, 1, 1]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights(
...     [1] * 10, [3, 2, 2])
[[1, 1, 1, 1], [1, 1, 1], [1, 1, 1]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights(
...     [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2], [1, 1])
[[1, 1, 1, 1, 1, 1, 2, 2], [2, 2, 2, 2]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights(
...     [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2], [1, 1, 1])
[[1, 1, 1, 1, 1, 1], [2, 2, 2], [2, 2, 2]]
```

Weights of parts of returned list equal *weights\_ratio* proportions with some rounding magic.

Returns list of lists.

### 29.1.70 sequencetools.partition\_sequence\_by\_restricted\_growth\_function

`sequencetools.partition_sequence_by_restricted_growth_function` (*sequence*,  
*restricted\_growth\_function*)

Partition *sequence* by *restricted\_growth\_function*:

```
>>> l = range(10)
>>> rgf = [1, 1, 2, 2, 1, 2, 3, 3, 2, 4]

>>> sequencetools.partition_sequence_by_restricted_growth_function(
...     l, rgf)
[[0, 1, 4], [2, 3, 5, 8], [6, 7], [9]]
```

Raise value error when *sequence* length does not equal *restricted\_growth\_function* length.

Returns list of lists.

### 29.1.71 sequencetools.partition\_sequence\_by\_sign\_of\_elements

`sequencetools.partition_sequence_by_sign_of_elements` (*sequence*, *sign*=[-1, 0, 1])

Partition *sequence* elements by sign:

```
>>> sequence = [0, 0, -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(
...     sequence))
[[0, 0], [-1, -1], [2, 3], [-5], [1, 2, 5], [-5, -6]]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(
...     sequence, sign=[-1]))
[0, 0, [-1, -1], 2, 3, [-5], 1, 2, 5, [-5, -6]]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(
...     sequence, sign=[0]))
[[0, 0], -1, -1, [2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(
...     sequence, sign=[1]))
[0, 0, -1, -1, [2, 3], -5, [1, 2, 5], -5, -6]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(
...     sequence, sign=[-1, 0]))
[[0, 0], [-1, -1], 2, 3, [-5], 1, 2, 5, [-5, -6]]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(
...     sequence, sign=[-1, 1]))
[0, 0, [-1, -1], [2, 3], [-5], [1, 2, 5], [-5, -6]]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(
...     sequence, sign=[0, 1]))
[[0, 0], -1, -1, [2, 3], -5, [1, 2, 5], -5, -6]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(
...     sequence, sign=[-1, 0, 1]))
[[0, 0], [-1, -1], [2, 3], [-5], [1, 2, 5], [-5, -6]]
```

When -1 in sign, group negative elements.

When 0 in sign, group 0 elements.

When 1 in sign, group positive elements.

Returns list of tuples of *sequence* element references.

### 29.1.72 sequencetools.partition\_sequence\_by\_value\_of\_elements

`sequencetools.partition_sequence_by_value_of_elements(sequence)`

Group *sequence* elements by value of elements:

```
>>> sequence = [0, 0, -1, -1, 2, 3, -5, 1, 1, 5, -5]
```

```
>>> sequencetools.partition_sequence_by_value_of_elements(sequence)
[(0, 0), (-1, -1), (2,), (3,), (-5,), (1, 1), (5,), (-5,)]
```

Returns list of tuples of *sequence* element references.

### 29.1.73 sequencetools.partition\_sequence\_by\_weights\_at\_least

`sequencetools.partition_sequence_by_weights_at_least(sequence, weights, cyclic=False, overhang=False)`

Partition *sequence* by *weights* at least.

```
>>> sequence = [3, 3, 3, 3, 4, 4, 4, 4, 5, 5]
```

**Example 1.** Partition sequence once by weights at least without overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_least(
...     sequence, [10, 4], cyclic=False, overhang=False)
[[3, 3, 3, 3], [4]]
```

**Example 2.** Partition sequence once by weights at least with overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_least(
...     sequence, [10, 4], cyclic=False, overhang=True)
[[3, 3, 3, 3], [4], [4, 4, 4, 5, 5]]
```

**Example 3.** Partition sequence cyclically by weights at least without overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_least(
...     sequence, [10, 4], cyclic=True, overhang=False)
[[3, 3, 3, 3], [4], [4, 4, 4], [5]]
```

**Example 4.** Partition sequence cyclically by weights at least with overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_least(
...     sequence, [10, 4], cyclic=True, overhang=True)
[[3, 3, 3, 3], [4], [4, 4, 4], [5], [5]]
```

Returns list of sequence objects.

### 29.1.74 sequencetools.partition\_sequence\_by\_weights\_at\_most

`sequencetools.partition_sequence_by_weights_at_most(sequence, weights, cyclic=False, overhang=False)`

Partition *sequence* by *weights* at most.

```
>>> sequence = [3, 3, 3, 3, 4, 4, 4, 4, 5, 5]
```

**Example 1.** Partition sequence once by weights at most without overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_most(
...     sequence,
...     [10, 4],
...     cyclic=False,
...     overhang=False,
... )
[[3, 3, 3], [3]]
```



**Example 2.** Partition sequence once by weights at most with overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_most(
...     sequence,
...     [10, 4],
...     cyclic=False,
...     overhang=True,
... )
[[3, 3, 3], [3], [4, 4, 4, 4, 5, 5]]
```

**Example 3.** Partition sequence cyclically by weights at most without overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_most(
...     sequence,
...     [10, 5],
...     cyclic=True,
...     overhang=False,
... )
[[3, 3, 3], [3], [4, 4], [4], [4, 5], [5]]
```

**Example 4.** Partition sequence cyclically by weights at most with overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_most(
...     sequence,
...     [10, 5],
...     cyclic=True,
...     overhang=True,
... )
[[3, 3, 3], [3], [4, 4], [4], [4, 5], [5]]
```

Returns list of sequence objects.

### 29.1.75 sequencetools.partition\_sequence\_by\_weights\_exactly

`sequencetools.partition_sequence_by_weights_exactly` (*sequence*, *weights*,  
*cyclic=False*, *overhang=False*)

Partition *sequence* by *weights* exactly.

```
>>> sequence = [3, 3, 3, 3, 4, 4, 4, 4, 5]
```

**Example 1.** Partition sequence once by weights exactly without overhang:

```
>>> sequencetools.partition_sequence_by_weights_exactly(
...     sequence,
...     [3, 9],
...     cyclic=False,
...     overhang=False,
... )
[[3], [3, 3, 3]]
```

**Example 2.** Partition sequence once by weights exactly with overhang:

```
>>> sequencetools.partition_sequence_by_weights_exactly(
...     sequence,
...     [3, 9],
...     cyclic=False,
...     overhang=True,
... )
[[3], [3, 3, 3], [4, 4, 4, 4, 5]]
```

**Example 3.** Partition sequence cyclically by weights exactly without overhang:

```
>>> sequencetools.partition_sequence_by_weights_exactly(
...     sequence,
...     [12],
...     cyclic=True,
...     overhang=False,
... )
[[3, 3, 3, 3], [4, 4, 4]]
```

**Example 4.** Partition sequence cyclically by weights exactly with overhang:

```
>>> sequencetools.partition_sequence_by_weights_exactly(  
...     sequence,  
...     [12],  
...     cyclic=True,  
...     overhang=True,  
... )  
[[3, 3, 3, 3], [4, 4, 4], [4, 5]]
```

Returns list sequence objects.

### 29.1.76 `sequencetools.partition_sequence_extended_to_counts`

`sequencetools.partition_sequence_extended_to_counts` (*sequence*, *counts*, *overhang=True*)

Partition sequence extended to counts.

**Example 1.** Partition sequence extended to counts with overhang:

```
>>> sequencetools.partition_sequence_extended_to_counts(  
...     (1, 2, 3, 4),  
...     (6, 6, 6),  
...     overhang=True,  
... )  
[(1, 2, 3, 4, 1, 2), (3, 4, 1, 2, 3, 4), (1, 2, 3, 4, 1, 2), (3, 4)]
```

**Example 2.** Partition sequence extended to counts without overhang:

```
>>> sequencetools.partition_sequence_extended_to_counts(  
...     (1, 2, 3, 4),  
...     (6, 6, 6),  
...     overhang=False,  
... )  
[(1, 2, 3, 4, 1, 2), (3, 4, 1, 2, 3, 4), (1, 2, 3, 4, 1, 2)]
```

Returns sequence of sequence objects.

### 29.1.77 `sequencetools.permute_sequence`

`sequencetools.permute_sequence` (*sequence*, *permutation*)

Permute *sequence* by *permutation*:

```
>>> sequencetools.permute_sequence([10, 11, 12, 13, 14, 15], [5, 4, 0, 1, 2, 3])  
[15, 14, 10, 11, 12, 13]
```

Returns newly constructed *sequence* object.

### 29.1.78 `sequencetools.remove_sequence_elements_at_indices`

`sequencetools.remove_sequence_elements_at_indices` (*sequence*, *indices*)

Remove *sequence* elements at *indices*:

```
>>> sequencetools.remove_sequence_elements_at_indices(range(20), [1, 16, 17, 18])  
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 19]
```

Ignore negative indices.

Returns list.

### 29.1.79 sequencetools.remove\_sequence\_elements\_at\_indices\_cyclically

`sequencetools.remove_sequence_elements_at_indices_cyclically` (*sequence*, *indices*, *period*, *offset=0*)

Remove *sequence* elements at *indices* mod *period* plus *offset*:

```
>>> sequencetools.remove_sequence_elements_at_indices_cyclically(range(20), [0, 1], 5, 3)
[0, 1, 2, 5, 6, 7, 10, 11, 12, 15, 16, 17]
```

Ignore negative indices.

Returns list.

### 29.1.80 sequencetools.remove\_subsequence\_of\_weight\_at\_index

`sequencetools.remove_subsequence_of_weight_at_index` (*sequence*, *weight*, *index*)

Remove subsequence of *weight* at *index*:

```
>>> sequence = (1, 1, 2, 3, 5, 5, 1, 2, 5, 5, 6)
```

```
>>> sequencetools.remove_subsequence_of_weight_at_index(sequence, 13, 4)
(1, 1, 2, 3, 5, 5, 6)
```

Returns newly constructed *sequence* object.

### 29.1.81 sequencetools.repeat\_runs\_in\_sequence\_to\_count

`sequencetools.repeat_runs_in_sequence_to_count` (*sequence*, *indicators*)

Repeat subruns in *sequence* according to *indicators*. The *indicators* input parameter must be a list of zero or more (start, length, count) triples. For every (start, length, count) indicator in *indicators*, the function copies `sequence[start:start+length]` and inserts count new copies of `sequence[start:start+length]` immediately after `sequence[start:start+length]` in *sequence*.

The function reads the value of count in every (start, length, count) triple not as the total number of occurrences of `sequence[start:start+length]` to appear in *sequence* after execution, but rather as the number of new occurrences of `sequence[start:start+length]` to appear in *sequence* after execution.

The function wraps newly created subruns in tuples. That is, this function returns output with one more level of nesting than given in input.

To insert 10 count of `sequence[:2]` at `sequence[2:2]`:

```
>>> sequencetools.repeat_runs_in_sequence_to_count(range(20), [(0, 2, 10)])
[0, 1, (0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1),
 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

To insert 5 count of `sequence[10:12]` at `sequence[12:12]` and then insert 5 count of `sequence[:2]` at `sequence[2:2]`:

```
>>> sequence = range(20)
```

```
>>> sequencetools.repeat_runs_in_sequence_to_count(sequence, [(0, 2, 5), (10, 2, 5)])
[0, 1, (0, 1, 0, 1, 0, 1, 0, 1, 0, 1), 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
 (10, 11, 10, 11, 10, 11, 10, 11, 10, 11), 12, 13, 14, 15, 16, 17, 18, 19]
```

---

**Note:** This function wraps around the end of *sequence* whenever `len(sequence) < start + length`.

---

To insert 2 count of `[18, 19, 0, 1]` at `sequence[2:2]`:

```
>>> sequencetools.repeat_runs_in_sequence_to_count(sequence, [(18, 4, 2)])
[0, 1, (18, 19, 0, 1, 18, 19, 0, 1), 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19]
```

To insert 2 count of [18, 19, 0, 1, 2, 3, 4] at sequence[4:4]:

```
>>> sequencetools.repeat_runs_in_sequence_to_count(sequence, [(18, 8, 2)])
[0, 1, 2, 3, 4, 5, (18, 19, 0, 1, 2, 3, 4, 5, 18, 19, 0, 1, 2, 3, 4, 5), 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

---

### Todo

Implement an optional *wrap* keyword to specify whether this function should wrap around the end of *sequence* whenever `len(sequence) < start + length` or not.

---

### Todo

Reimplement this function to return a generator.

---

Generalizations of this function would include functions to repeat subruns in *sequence* to not only a certain count, as implemented here, but to a certain length, weight or sum. That is, `sequencetools.repeat_subruns_to_length()`, `sequencetools.repeat_subruns_to_weight()` and `sequencetools.repeat_subruns_to_sum()`.

## 29.1.82 sequencetools.repeat\_sequence\_elements\_at\_indices

`sequencetools.repeat_sequence_elements_at_indices(sequence, indices, total)`

Repeat *sequence* elements at *indices* to *total* length:

```
>>> sequencetools.repeat_sequence_elements_at_indices(range(10), [6, 7, 8], 3)
[0, 1, 2, 3, 4, 5, [6, 6, 6], [7, 7, 7], [8, 8, 8], 9]
```

Returns list.

## 29.1.83 sequencetools.repeat\_sequence\_elements\_at\_indices\_cyclically

`sequencetools.repeat_sequence_elements_at_indices_cyclically(sequence, cycle_token, total)`

Repeat *sequence* elements at indices specified by *cycle\_token* to *total* length:

```
>>> sequencetools.repeat_sequence_elements_at_indices_cyclically(
...     range(10), (5, [1, 2]), 3)
[0, [1, 1, 1], [2, 2, 2], 3, 4, 5, [6, 6, 6], [7, 7, 7], 8, 9]
```

The *cycle\_token* may be a sieve:

```
>>> sieve = sievetools.Sieve.from_cycle_tokens((5, [1, 2]))
>>> sequencetools.repeat_sequence_elements_at_indices_cyclically(
...     range(10), sieve, 3)
[0, [1, 1, 1], [2, 2, 2], 3, 4, 5, [6, 6, 6], [7, 7, 7], 8, 9]
```

Returns list.

## 29.1.84 sequencetools.repeat\_sequence\_elements\_n\_times\_each

`sequencetools.repeat_sequence_elements_n_times_each(sequence, n)`

Repeat *sequence* elements *n* times each:

```
>>> sequencetools.repeat_sequence_elements_n_times_each((1, -1, 2, -3, 5, -5, 6), 2)
(1, 1, -1, -1, 2, 2, -3, -3, 5, 5, -5, -5, 6, 6)
```

Returns newly constructed *sequence* object with copied *sequence* elements.

### 29.1.85 sequencetools.repeat\_sequence\_n\_times

`sequencetools.repeat_sequence_n_times(sequence, n)`

Repeat *sequence* *n* times:

```
>>> sequencetools.repeat_sequence_n_times((1, 2, 3, 4, 5), 3)
(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

Repeat *sequence* 0 times:

```
>>> sequencetools.repeat_sequence_n_times((1, 2, 3, 4, 5), 0)
()
```

Returns newly constructed *sequence* object of copied *sequence* elements.

### 29.1.86 sequencetools.repeat\_sequence\_to\_length

`sequencetools.repeat_sequence_to_length(sequence, length, start=0)`

Repeat *sequence* to nonnegative integer *length*:

```
>>> sequencetools.repeat_sequence_to_length(range(5), 11)
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0]
```

Repeat *sequence* to nonnegative integer *length* from *start*:

```
>>> sequencetools.repeat_sequence_to_length(range(5), 11, start=2)
[2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2]
```

Returns newly constructed *sequence* object.

### 29.1.87 sequencetools.repeat\_sequence\_to\_weight\_at\_least

`sequencetools.repeat_sequence_to_weight_at_least(sequence, weight)`

Repeat *sequence* to *weight* at least:

```
>>> sequencetools.repeat_sequence_to_weight_at_least((5, -5, -5), 23)
(5, -5, -5, 5, -5)
```

Returns newly constructed *sequence* object.

### 29.1.88 sequencetools.repeat\_sequence\_to\_weight\_at\_most

`sequencetools.repeat_sequence_to_weight_at_most(sequence, weight)`

Repeat *sequence* to *weight* at most:

```
>>> sequencetools.repeat_sequence_to_weight_at_most((5, -5, -5), 23)
(5, -5, -5, 5)
```

Returns newly constructed *sequence* object.

### 29.1.89 sequencetools.repeat\_sequence\_to\_weight\_exactly

`sequencetools.repeat_sequence_to_weight_exactly(sequence, weight)`

Repeat *sequence* to *weight* exactly:

```
>>> sequencetools.repeat_sequence_to_weight_exactly((5, -5, -5), 23)
(5, -5, -5, 5, -3)
```

Returns newly constructed *sequence* object.

### 29.1.90 sequencetools.replace\_sequence\_elements\_cyclically\_with\_new\_material

`sequencetools.replace_sequence_elements_cyclically_with_new_material(sequence, indices, new_material)`

Replace *sequence* elements cyclically at *indices* with *new\_material*:

```
>>> sequencetools.replace_sequence_elements_cyclically_with_new_material(
... range(20), ([0], 2), (['A', 'B'], 3))
['A', 1, 'B', 3, 4, 5, 'A', 7, 'B', 9, 10, 11, 'A', 13, 'B', 15, 16, 17, 'A', 19]
```

```
>>> sequencetools.replace_sequence_elements_cyclically_with_new_material(
... range(20), ([0], 2), (['*'], 1))
['*', 1, '*', 3, '*', 5, '*', 7, '*', 9, '*', 11, '*', 13, '*', 15, '*', 17, '*', 19]
```

```
>>> sequencetools.replace_sequence_elements_cyclically_with_new_material(
... range(20), ([0], 2), (['A', 'B', 'C', 'D'], None))
['A', 1, 'B', 3, 'C', 5, 'D', 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
>>> sequencetools.replace_sequence_elements_cyclically_with_new_material(
... range(20), ([0, 1, 8, 13], None), (['A', 'B', 'C', 'D'], None))
['A', 'B', 2, 3, 4, 5, 6, 7, 'C', 9, 10, 11, 12, 'D', 14, 15, 16, 17, 18, 19]
```

Raise type error when *sequence* not a list.

Returns new list.

### 29.1.91 sequencetools.retain\_sequence\_elements\_at\_indices

`sequencetools.retain_sequence_elements_at_indices(sequence, indices)`

Retain *sequence* elements at *indices*:

```
>>> sequencetools.retain_sequence_elements_at_indices(range(20), [1, 16, 17, 18])
[1, 16, 17, 18]
```

Returns sequence elements in the order they appear in *sequence*.

Ignore negative indices.

Returns list.

### 29.1.92 sequencetools.retain\_sequence\_elements\_at\_indices\_cyclically

`sequencetools.retain_sequence_elements_at_indices_cyclically(sequence, indices, period, offset=0)`

Retain *sequence* elements at *indices* mod *period* plus *offset*:

```
>>> sequencetools.retain_sequence_elements_at_indices_cyclically(range(20), [0, 1], 5, 3)
[3, 4, 8, 9, 13, 14, 18, 19]
```

Ignore negative values in *indices*.

Returns list.

### 29.1.93 sequencetools.reverse\_sequence

`sequencetools.reverse_sequence(sequence)`

Reverse *sequence*:

```
>>> sequencetools.reverse_sequence((1, 2, 3, 4, 5))
(5, 4, 3, 2, 1)
```

Returns new *sequence* object.

### 29.1.94 sequencetools.reverse\_sequence\_elements

`sequencetools.reverse_sequence_elements(sequence)`

Reverse *sequence* elements:

```
>>> sequencetools.reverse_sequence_elements([1, (2, 3, 4), 5, (6, 7)])
[1, (4, 3, 2), 5, (7, 6)]
```

Returns new *sequence* object.

### 29.1.95 sequencetools.rotate\_sequence

`sequencetools.rotate_sequence(sequence, n)`

Rotate *sequence* to the right:

```
>>> sequencetools.rotate_sequence(range(10), 4)
[6, 7, 8, 9, 0, 1, 2, 3, 4, 5]
```

Rotate *sequence* to the left:

```
>>> sequencetools.rotate_sequence(range(10), -3)
[3, 4, 5, 6, 7, 8, 9, 0, 1, 2]
```

Rotate *sequence* neither to the right nor the left:

```
>>> sequencetools.rotate_sequence(range(10), 0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Returns newly created *sequence* object.

### 29.1.96 sequencetools.splice\_new\_elements\_between\_sequence\_elements

`sequencetools.splice_new_elements_between_sequence_elements(sequence, new_elements, overhang=(0, 0))`

Splice copies of *new\_elements* between each of the elements of *sequence*:

```
>>> sequence = [0, 1, 2, 3, 4]
>>> new_elements = ['A', 'B']
```

```
>>> sequencetools.splice_new_elements_between_sequence_elements(sequence, new_elements)
[0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4]
```

Splice copies of *new\_elements* between each of the elements of *sequence* and after the last element of *sequence*:

```
>>> sequencetools.splice_new_elements_between_sequence_elements(
...     sequence, new_elements, overhang=(0, 1))
[0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4, 'A', 'B']
```

Splice copies of *new\_elements* before the first element of *sequence* and between each of the other elements of *sequence*:

```
>>> sequencetools.splice_new_elements_between_sequence_elements(
...     sequence, new_elements, overhang=(1, 0))
['A', 'B', 0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4]
```

Splice copies of *new\_elements* before the first element of *sequence*, after the last element of *sequence* and between each of the other elements of *sequence*:

```
>>> sequencetools.splice_new_elements_between_sequence_elements(
...     sequence, new_elements, overhang=(1, 1))
['A', 'B', 0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4, 'A', 'B']
```

Returns newly constructed list.

### 29.1.97 sequencetools.split\_sequence\_by\_weights

`sequencetools.split_sequence_by_weights(sequence, weights, cyclic=False, overhang=False)`

Split sequence by weights.

**Example 1.** Split sequence cyclically by weights with overhang:

```
>>> sequencetools.split_sequence_by_weights(
...     (10, -10, 10, -10),
...     (3, 15, 3),
...     cyclic=True,
...     overhang=True,
...     )
[(3,), (7, -8), (-2, 1), (3,), (6, -9), (-1,)]
```

**Example 2.** Split sequence cyclically by weights without overhang:

```
>>> sequencetools.split_sequence_by_weights(
...     (10, -10, 10, -10),
...     (3, 15, 3),
...     cyclic=True,
...     overhang=False,
...     )
[(3,), (7, -8), (-2, 1), (3,), (6, -9)]
```

**Example 3.** Split sequence once by weights with overhang:

```
>>> sequencetools.split_sequence_by_weights(
...     (10, -10, 10, -10),
...     (3, 15, 3),
...     cyclic=False,
...     overhang=True,
...     )
[(3,), (7, -8), (-2, 1), (9, -10)]
```

**Example 4.** Split sequence once by weights without overhang:

```
>>> sequencetools.split_sequence_by_weights(
...     (10, -10, 10, -10),
...     (3, 15, 3),
...     cyclic=False,
...     overhang=False,
...     )
[(3,), (7, -8), (-2, 1)]
```

Returns list of sequence types.



### 29.1.98 sequencetools.split\_sequence\_extended\_to\_weights

`sequencetools.split_sequence_extended_to_weights` (*sequence*, *weights*, *overhang=True*)

Split sequence extended to weights.

**Example 1.** Split sequence extended to weights with overhang:

```
>>> sequencetools.split_sequence_extended_to_weights(
...     [1, 2, 3, 4, 5], [7, 7, 7], overhang=True)
[[1, 2, 3, 1], [3, 4], [1, 1, 2, 3], [4, 5]]
```

**Example 2.** Split sequence extended to weights without overhang:

```
>>> sequencetools.split_sequence_extended_to_weights(
...     [1, 2, 3, 4, 5], [7, 7, 7], overhang=False)
[[1, 2, 3, 1], [3, 4], [1, 1, 2, 3]]
```

Returns sequence of sequence objects.

### 29.1.99 sequencetools.sum\_consecutive\_sequence\_elements\_by\_sign

`sequencetools.sum_consecutive_sequence_elements_by_sign` (*sequence*, *sign*=[-1, 0, 1])

Sum consecutive *sequence* elements by *sign*:

```
>>> sequence = [0, 0, -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence)
[0, -2, 5, -5, 8, -11]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[-1])
[0, 0, -2, 2, 3, -5, 1, 2, 5, -11]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[0])
[0, -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[1])
[0, 0, -1, -1, 5, -5, 8, -5, -6]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[-1, 0])
[0, -2, 2, 3, -5, 1, 2, 5, -11]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[-1, 1])
[0, 0, -2, 5, -5, 8, -11]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[0, 1])
[0, -1, -1, 5, -5, 8, -5, -6]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[-1, 0, 1])
[0, -2, 5, -5, 8, -11]
```

When -1 in *sign*, sum consecutive negative elements.

When 0 in *sign*, sum consecutive 0 elements.

When 1 in *sign*, sum consecutive positive elements.

Returns list.

### 29.1.100 sequencetools.sum\_sequence\_elements\_at\_indices

`sequencetools.sum_sequence_elements_at_indices` (*sequence*, *pairs*, *period=None*, *overhang=True*)

Sum *sequence* elements at indices according to *pairs*:

```
>>> sequencetools.sum_sequence_elements_at_indices(range(10), [(0, 3)])
[3, 3, 4, 5, 6, 7, 8, 9]
```

Sum *sequence* elements cyclically at indices according to *pairs* and *period*:

```
>>> sequencetools.sum_sequence_elements_at_indices(range(10), [(0, 3)], period=4)
[3, 3, 15, 7, 17]
```

Sum *sequence* elements cyclically at indices according to *pairs* and *period* and do not return incomplete final sum:

```
>>> sequencetools.sum_sequence_elements_at_indices(
...     range(10), [(0, 3)], period=4, overhang=False)
[3, 3, 15, 7]
```

Replace `sequence[i:i+count]` with `sum(sequence[i:i+count])` for each `(i, count)` in *pairs*.

Indices in *pairs* must be less than *period* when *period* is not none.

Returns new list.

### 29.1.101 sequencetools.truncate\_runs\_in\_sequence

`sequencetools.truncate_runs_in_sequence(sequence)`

Truncate subruns of like elements in *sequence* to length 1:

```
>>> sequencetools.truncate_runs_in_sequence([1, 1, 2, 3, 3, 3, 9, 4, 4, 4])
[1, 2, 3, 9, 4]
```

Returns empty list when *sequence* is empty:

```
>>> sequencetools.truncate_runs_in_sequence([])
[]
```

Raise type error when *sequence* is not a list.

Returns new list.

### 29.1.102 sequencetools.truncate\_sequence\_to\_sum

`sequencetools.truncate_sequence_to_sum(sequence, target_sum)`

Truncate *sequence* to *target\_sum*:

```
>>> sequence = [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
```

```
>>> for n in range(10):
...     print n, sequencetools.truncate_sequence_to_sum(sequence, n)
...
0 []
1 [-1, 2]
2 [-1, 2, -3, 4]
3 [-1, 2, -3, 4, -5, 6]
4 [-1, 2, -3, 4, -5, 6, -7, 8]
5 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
6 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
7 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
8 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
9 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
```

Returns empty list when *target\_sum* is 0:

```
>>> sequencetools.truncate_sequence_to_sum([1, 2, 3, 4, 5], 0)
[]
```

Raise type error when *sequence* is not a list.

Raise value error on negative *target\_sum*.

Returns new list.

### 29.1.103 sequencetools.truncate\_sequence\_to\_weight

`sequencetools.truncate_sequence_to_weight(sequence, weight)`

Truncate *sequence* to *weight*:

```
>>> l = [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
>>> for x in range(10):
...     print x, sequencetools.truncate_sequence_to_weight(l, x)
...
0 []
1 [-1]
2 [-1, 1]
3 [-1, 2]
4 [-1, 2, -1]
5 [-1, 2, -2]
6 [-1, 2, -3]
7 [-1, 2, -3, 1]
8 [-1, 2, -3, 2]
9 [-1, 2, -3, 3]
```

Returns empty list when *weight* is 0:

```
>>> sequencetools.truncate_sequence_to_weight([1, 2, 3, 4, 5], 0)
[]
```

Raise type error when *sequence* is not a list.

Raise value error on negative *weight*.

Returns new list.

### 29.1.104 sequencetools.yield\_all\_combinations\_of\_sequence\_elements

`sequencetools.yield_all_combinations_of_sequence_elements(sequence, min_length=None, max_length=None)`

Yield all combinations of *sequence* in binary string order:

```
>>> list(sequencetools.yield_all_combinations_of_sequence_elements(
...     [1, 2, 3, 4]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3], [4], [1, 4],
[2, 4], [1, 2, 4], [3, 4], [1, 3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

Yield all combinations of *sequence* greater than or equal to *min\_length* in binary string order:

```
>>> list(sequencetools.yield_all_combinations_of_sequence_elements(
...     [1, 2, 3, 4], min_length=3))
[[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

Yield all combinations of *sequence* less than or equal to *max\_length* in binary string order:

```
>>> list(sequencetools.yield_all_combinations_of_sequence_elements(
...     [1, 2, 3, 4], max_length=2))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [4], [1, 4], [2, 4], [3, 4]]
```

Yield all combinations of *sequence* greater than or equal to *min\_length* and less than or equal to *max\_length* in lex order:

```
>>> list(sequencetools.yield_all_combinations_of_sequence_elements(
...     [1, 2, 3, 4], min_length=2, max_length=2))
[[1, 2], [1, 3], [2, 3], [1, 4], [2, 4], [3, 4]]
```

Returns generator of newly created *sequence* objects.

### 29.1.105 `sequencetools.yield_all_k_ary_sequences_of_length`

`sequencetools.yield_all_k_ary_sequences_of_length(k, length)`

Generate all  $k$ -ary sequences of *length*:

```
>>> for sequence in sequencetools.yield_all_k_ary_sequences_of_length(2, 3):
...     sequence
...
(0, 0, 0)
(0, 0, 1)
(0, 1, 0)
(0, 1, 1)
(1, 0, 0)
(1, 0, 1)
(1, 1, 0)
(1, 1, 1)
```

Returns generator of tuples.

### 29.1.106 `sequencetools.yield_all_pairs_between_sequences`

`sequencetools.yield_all_pairs_between_sequences(l, m)`

Yield all pairs between sequences *l* and *m*:

```
>>> for pair in sequencetools.yield_all_pairs_between_sequences([1, 2, 3], [4, 5]):
...     pair
...
(1, 4)
(1, 5)
(2, 4)
(2, 5)
(3, 4)
(3, 5)
```

Returns pair generator.

### 29.1.107 `sequencetools.yield_all_partitions_of_sequence`

`sequencetools.yield_all_partitions_of_sequence(sequence)`

Yield all partitions of *sequence*:

```
>>> for partition in sequencetools.yield_all_partitions_of_sequence([0, 1, 2, 3]):
...     partition
...
[[0, 1, 2, 3]]
[[0, 1, 2], [3]]
[[0, 1], [2, 3]]
[[0, 1], [2], [3]]
[[0], [1, 2, 3]]
[[0], [1, 2], [3]]
[[0], [1], [2, 3]]
[[0], [1], [2], [3]]
```

Returns generator of newly created lists.

### 29.1.108 `sequencetools.yield_all_permutations_of_sequence`

`sequencetools.yield_all_permutations_of_sequence(sequence)`

Yield all permutations of *sequence* in lex order:

```
>>> list(sequencetools.yield_all_permutations_of_sequence((1, 2, 3)))
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

Returns generator of *sequence* objects.

### 29.1.109 sequencetools.yield\_all\_permutations\_of\_sequence\_in\_orbit

`sequencetools.yield_all_permutations_of_sequence_in_orbit` (*sequence*, *permutation*)

Yield all permutations of *sequence* in orbit of *permutation* in lex order:

```
>>> list(sequencetools.yield_all_permutations_of_sequence_in_orbit(
...     (1, 2, 3, 4), [1, 2, 3, 0]))
[(1, 2, 3, 4), (2, 3, 4, 1), (3, 4, 1, 2), (4, 1, 2, 3)]
```

Returns generator of *sequence* objects.

### 29.1.110 sequencetools.yield\_all\_restricted\_growth\_functions\_of\_length

`sequencetools.yield_all_restricted_growth_functions_of_length` (*length*)

Generate all restricted growth functions of *length* in lex order:

```
>>> for rgf in sequencetools.yield_all_restricted_growth_functions_of_length(4):
...     rgf
...
(1, 1, 1, 1)
(1, 1, 1, 2)
(1, 1, 2, 1)
(1, 1, 2, 2)
(1, 1, 2, 3)
(1, 2, 1, 1)
(1, 2, 1, 2)
(1, 2, 1, 3)
(1, 2, 2, 1)
(1, 2, 2, 2)
(1, 2, 2, 3)
(1, 2, 3, 1)
(1, 2, 3, 2)
(1, 2, 3, 3)
(1, 2, 3, 4)
```

Returns generator of tuples.

### 29.1.111 sequencetools.yield\_all\_rotations\_of\_sequence

`sequencetools.yield_all_rotations_of_sequence` (*sequence*, *n=1*)

Yield all *n*-rotations of *sequence* up to identity:

```
>>> list(sequencetools.yield_all_rotations_of_sequence([1, 2, 3, 4], -1))
[[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]]
```

Returns generator of *sequence* objects.

### 29.1.112 sequencetools.yield\_all\_set\_partitions\_of\_sequence

`sequencetools.yield_all_set_partitions_of_sequence` (*sequence*)

Yield all set partitions of *sequence* in restricted growth function order:

```
>>> for set_partition in sequencetools.yield_all_set_partitions_of_sequence(
...     [21, 22, 23, 24]):
...     set_partition
...
```

```
[21, 22, 23, 24]]
[[21, 22, 23], [24]]
[[21, 22, 24], [23]]
[[21, 22], [23, 24]]
[[21, 22], [23], [24]]
[[21, 23, 24], [22]]
[[21, 23], [22, 24]]
[[21, 23], [22], [24]]
[[21, 24], [22, 23]]
[[21], [22, 23, 24]]
[[21], [22, 23], [24]]
[[21, 24], [22], [23]]
[[21], [22, 24], [23]]
[[21], [22], [23, 24]]
[[21], [22], [23], [24]]
```

Returns generator of list of lists.

### 29.1.113 `sequencetools.yield_all_subsequences_of_sequence`

`sequencetools.yield_all_subsequences_of_sequence` (*sequence*, *min\_length=0*,  
*max\_length=None*)

Yield all subsequences of *sequence* in lex order:

```
>>> list(sequencetools.yield_all_subsequences_of_sequence([0, 1, 2]))
[[], [0], [0, 1], [0, 1, 2], [1], [1, 2], [2]]
```

Yield all subsequences of *sequence* greater than or equal to *min\_length* in lex order:

```
>>> list(sequencetools.yield_all_subsequences_of_sequence(
...     [0, 1, 2, 3, 4], min_length=3))
[[0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4], [1, 2, 3], [1, 2, 3, 4], [2, 3, 4]]
```

Yield all subsequences of *sequence* less than or equal to *max\_length* in lex order:

```
>>> for subsequence in sequencetools.yield_all_subsequences_of_sequence(
...     [0, 1, 2, 3, 4], max_length=3):
...     subsequence
[]
[0]
[0, 1]
[0, 1, 2]
[1]
[1, 2]
[1, 2, 3]
[2]
[2, 3]
[2, 3, 4]
[3]
[3, 4]
[4]
```

Yield all subsequences of *sequence* greater than or equal to *min\_length* and less than or equal to *max\_length* in lex order:

```
>>> list(sequencetools.yield_all_subsequences_of_sequence(
...     [0, 1, 2, 3, 4], min_length=3, max_length=3))
[[0, 1, 2], [1, 2, 3], [2, 3, 4]]
```

Returns generator of newly created *sequence* slices.

### 29.1.114 `sequencetools.yield_all_unordered_pairs_of_sequence`

`sequencetools.yield_all_unordered_pairs_of_sequence` (*sequence*)

Yield all unordered pairs of *sequence*:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([1, 2, 3, 4]))
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

Yield all unordered pairs of length-1 *sequence*:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([1]))
[]
```

Yield all unordered pairs of empty *sequence*:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([]))
[]
```

Yield all unordered pairs of *sequence* with duplicate elements:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([1, 1, 1]))
[(1, 1), (1, 1), (1, 1)]
```

Pairs are tuples instead of sets to accommodate duplicate *sequence* elements.

Returns generator.

### 29.1.115 sequencetools.yield\_outer\_product\_of\_sequences

`sequencetools.yield_outer_product_of_sequences` (*sequences*)

Yield outer product of *sequences*:

```
>>> list(sequencetools.yield_outer_product_of_sequences(
...     [[1, 2, 3], ['a', 'b']]))
[[1, 'a'], [1, 'b'], [2, 'a'], [2, 'b'], [3, 'a'], [3, 'b']]
```

```
>>> list(sequencetools.yield_outer_product_of_sequences(
...     [[1, 2, 3], ['a', 'b'], ['X', 'Y']]))
[[1, 'a', 'X'], [1, 'a', 'Y'], [1, 'b', 'X'], [1, 'b', 'Y'],
 [2, 'a', 'X'], [2, 'a', 'Y'], [2, 'b', 'X'], [2, 'b', 'Y'],
 [3, 'a', 'X'], [3, 'a', 'Y'], [3, 'b', 'X'], [3, 'b', 'Y']]
```

```
>>> list(sequencetools.yield_outer_product_of_sequences(
...     [[1, 2, 3], [4, 5], [6, 7, 8]]))
[[1, 4, 6], [1, 4, 7], [1, 4, 8], [1, 5, 6], [1, 5, 7], [1, 5, 8],
 [2, 4, 6], [2, 4, 7], [2, 4, 8], [2, 5, 6], [2, 5, 7], [2, 5, 8],
 [3, 4, 6], [3, 4, 7], [3, 4, 8], [3, 5, 6], [3, 5, 7], [3, 5, 8]]
```

Returns generator.

### 29.1.116 sequencetools.zip\_sequences\_cyclically

`sequencetools.zip_sequences_cyclically` (*\*sequences*)

Zip *sequences* cyclically:

```
>>> sequencetools.zip_sequences_cyclically([1, 2, 3], ['a', 'b'])
[(1, 'a'), (2, 'b'), (3, 'a')]
```

Arbitrary number of input sequences now allowed.

```
>>> sequencetools.zip_sequences_cyclically([10, 11, 12], [20, 21], [30, 31, 32, 33])
[(10, 20, 30), (11, 21, 31), (12, 20, 32), (10, 21, 33)]
```

Cycle over the elements of the sequences of shorter length.

Returns list of length equal to sequence of greatest length in *sequences*.

### 29.1.117 `sequencetools.zip_sequences_without_truncation`

`sequencetools.zip_sequences_without_truncation(*sequences)`

Zip *sequences* nontruncating:

```
>>> sequencetools.zip_sequences_without_truncation(  
...     [1, 2, 3, 4], [11, 12, 13], [21, 22, 23])  
[(1, 11, 21), (2, 12, 22), (3, 13, 23), (4,)]
```

Lengths of the tuples returned may differ but will always be greater than or equal to 1.

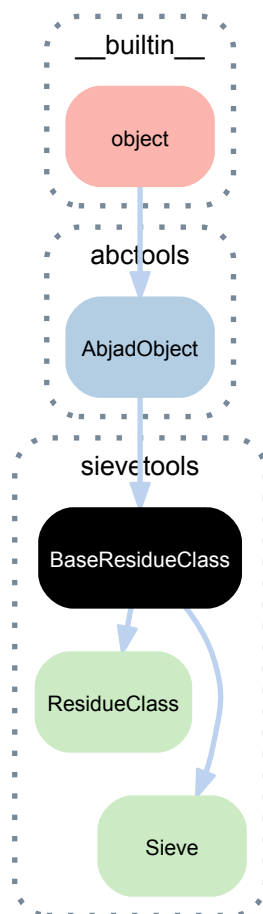
Returns list of tuples.



# SIEVETOOLS

## 30.1 Concrete classes

### 30.1.1 `sievetools.BaseResidueClass`



**class** `sievetools.BaseResidueClass`  
Abstract base class for `ResidueClass` and `Sieve`.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Special methods

`BaseResidueClass.__and__(arg)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`BaseResidueClass.__or__(arg)`

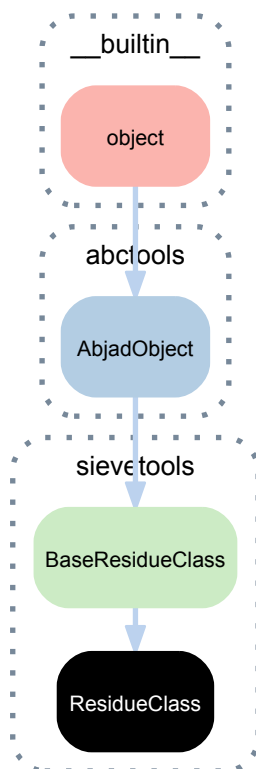
`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

`BaseResidueClass.__xor__(arg)`

### 30.1.2 sievetools.ResidueClass



**class** `sievetools.ResidueClass(*args)`

Residue class (or congruence class).

Residue classes form the basis of Xenakis sieves. They can be used to make any complex periodic integer or boolean sequence as a combination of simple periodic sequences.

**Example.** From the opening of Xenakis's *Psappha* for solo percussion:

```
>>> RC = sievetools.ResidueClass
```

```
>>> s1 = (RC(8, 0) | RC(8, 1) | RC(8, 7)) & (RC(5, 1) | RC(5, 3))
```

```
>>> s2 = (RC(8, 0) | RC(8, 1) | RC(8, 2)) & RC(5, 0)
```

```
>>> s3 = RC(8, 3)
```

```
>>> s4 = RC(8, 4)
>>> s5 = (RC(8, 5) | RC(8, 6)) & (RC(5, 2) | RC(5, 3) | RC(5, 4))
>>> s6 = (RC(8, 1) & RC(5, 2))
>>> s7 = (RC(8, 6) & RC(5, 1))
```

```
>>> y = s1 | s2 | s3 | s4 | s5 | s6 | s7
```

```
>>> y.get_congruent_bases(40)
[0, 1, 3, 4, 6, 8, 10, 11, 12, 13, 14, 16, 17, 19, 20, 22,
 23, 25, 27, 28, 29, 31, 33, 35, 36, 37, 38, 40]
```

```
>>> y.get_boolean_train(40)
[1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1,
 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0]
```

Returns residue class.

## Bases

- `sievetools.BaseResidueClass`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ResidueClass.modulo`  
Period of residue class.

`ResidueClass.residue`  
Residue of residue class.

## Methods

`ResidueClass.get_boolean_train(*min_max)`

Returns a boolean train with 0s mapped to the integers that are not congruent bases of the residue class and 1s mapped to those that are.

The method takes one or two integer arguments. If only one is given, it is taken as the max range and the min is assumed to be 0.

**Example:**

```
>>> r = RC(3, 0)
>>> r.get_boolean_train(6)
[1, 0, 0, 1, 0, 0]
```

```
>>> r.get_congruent_bases(-6, 6)
[-6, -3, 0, 3, 6]
```

Returns list.

`ResidueClass.get_congruent_bases(*min_max)`

Returns all the congruent bases of this residue class within the given range.

The method takes one or two integer arguments. If only one it given, it is taken as the max range and the min is assumed to be 0.

**Example:**

```
>>> r = RC(3, 0)
>>> r.get_congruent_bases(6)
[0, 3, 6]
```

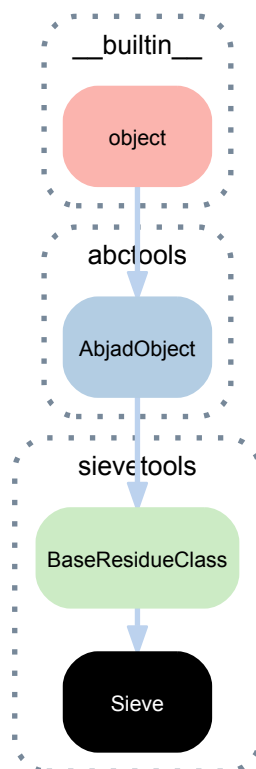
```
>>> r.get_congruent_bases(-6, 6)
[-6, -3, 0, 3, 6]
```

Returns list.

## Special methods

```
(BaseResidueClass).__and__(arg)
ResidueClass.__eq__(exp)
ResidueClass.__ge__(expr)
ResidueClass.__gt__(expr)
ResidueClass.__le__(expr)
ResidueClass.__lt__(expr)
ResidueClass.__ne__(expr)
(BaseResidueClass).__or__(arg)
ResidueClass.__repr__()
(BaseResidueClass).__xor__(arg)
```

## 30.1.3 sievetools.Sieve



```
class sievetools.Sieve(rcs, logical_operator='or')
```

## Bases

- `sievetools.BaseResidueClass`
- `abctools.AbjadObject`

- `__builtin__.object`

## Read-only properties

`Sieve.logical_operator`

Residue class expression logical operator.

`Sieve.period`

Residue class expression period.

`Sieve.rcs`

Residue class expression residue classes.

`Sieve.representative_boolean_train`

Residue class expression representative boolean train.

`Sieve.representative_congruent_bases`

Residue class expression representative congruent bases.

## Methods

`Sieve.get_boolean_train(*min_max)`

Returns a boolean train with 0s mapped to the integers that are not congruent bases of the residue class expression and 1s mapped to those that are.

The method takes one or two integer arguments. If only one is given, it is taken as the max range and min is assumed to be 0.

**Example:**

```
>>> from abjad.tools.sievetools import ResidueClass
```

```
>>> e = ResidueClass(3, 0) | ResidueClass(2, 0)
>>> e.get_boolean_train(6)
[1, 0, 1, 1, 1, 0]
>>> e.get_congruent_bases(-6, 6)
[-6, -4, -3, -2, 0, 2, 3, 4, 6]
```

Returns list.

`Sieve.get_congruent_bases(*min_max)`

Returns all the congruent bases of this residue class expression within the given range.

The method takes one or two integer arguments. If only one is given, it is taken as the max range and min is assumed to be 0.

**Example:**

```
>>> e = ResidueClass(3, 0) | ResidueClass(2, 0)
>>> e.get_congruent_bases(6)
[0, 2, 3, 4, 6]
>>> e.get_congruent_bases(-6, 6)
[-6, -4, -3, -2, 0, 2, 3, 4, 6]
```

Returns list.

`Sieve.is_congruent_base(integer)`

## Static methods

`Sieve.from_cycle_tokens(*cycle_tokens)`

Make Xenakis sieve from *cycle\_tokens*:

```
>>> cycle_token_1 = (6, [0, 4, 5])
>>> cycle_token_2 = (10, [0, 1, 2], 6)
>>> cycle_tokens = [cycle_token_1, cycle_token_2]
```

```
>>> sievetools.Sieve.from_cycle_tokens(*cycle_tokens)
{ResidueClass(6, 0) | ResidueClass(6, 4) | ResidueClass(6, 5) |
ResidueClass(10, 6) | ResidueClass(10, 7) | ResidueClass(10, 8)}
```

Cycle token comprises *modulo*, *residues* and optional *offset*.

## Special methods

(BaseResidueClass) .**\_\_and\_\_**(arg)

(AbjadObject) .**\_\_eq\_\_**(expr)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(expr)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(BaseResidueClass) .**\_\_or\_\_**(arg)

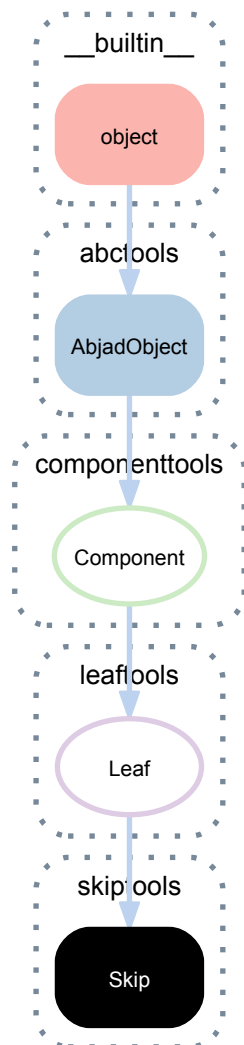
Sieve.**\_\_repr\_\_**()

(BaseResidueClass) .**\_\_xor\_\_**(arg)

# SKIPTOOLS

## 31.1 Concrete classes

### 31.1.1 skiptools.Skip



```
class skiptools.Skip(*args, **kwargs)  
    A LilyPond skip.
```

**Example.**

```
>>> skip = skiptools.Skip((3, 16))
>>> skip
Skip('s8.')
```

## Bases

- `leaftools.Leaf`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Component).lilypond_format`  
Lilypond format of component.

Returns string.

`(Component).override`  
LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Component).set`  
LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

`(Component).storage_format`  
Storage format of component.

Returns string.

## Read/write properties

`(Leaf).lilypond_duration_multiplier`  
LilyPond duration multiplier.

Set to positive multiplier or none.

Returns positive multiplier or none.

`(Leaf).written_duration`  
Written duration of leaf.

Set to duration.

Returns duration.

`(Leaf).written_pitch_indication_is_at_sounding_pitch`  
Returns true when written pitch is at sounding pitch. Returns false when written pitch is transposed.

`(Leaf).written_pitch_indication_is_nonsemantic`  
Returns true when pitch is nonsemantic. Returns false otherwise.

Set to true when using leaves only graphically.

Setting this value to true sets sounding pitch indicator to false.



## Methods

(Component) .**select** (*sequential=False*)  
 Selects component.  
 Returns component selection when *sequential* is false.  
 Returns sequential selection when *sequential* is true.

## Special methods

(Component) .**\_\_copy\_\_** (\*args)  
 Copies component with marks but without children of component or spanners attached to component.  
 Returns new component.

(AbjadObject) .**\_\_eq\_\_** (expr)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(Component) .**\_\_mul\_\_** (n)  
 Copies component *n* times and detaches spanners.  
 Returns list of new components.

(AbjadObject) .**\_\_ne\_\_** (expr)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Leaf) .**\_\_repr\_\_** ()  
 Interpreter representation of leaf.  
 Returns string.

(Component) .**\_\_rmul\_\_** (n)  
 Copies component *n* times and detach spanners.  
 Returns list of new components.

(Leaf) .**\_\_str\_\_** ()  
 String representation of leaf.  
 Returns string.

## 31.2 Functions

### 31.2.1 skiptools.make\_repeated\_skips\_from\_time\_signatures

skiptools.**make\_repeated\_skips\_from\_time\_signatures** (*time\_signatures*)  
 Make repeated skips from *time\_signatures*:

```
skiptools.make_repeated_skips_from_time_signatures([(2, 8), (3, 32)])
[Selection(Skip('s8'), Skip('s8')), Selection(Skip('s32'), Skip('s32'), Skip('s32'))]
```

Returns two-dimensional list of newly constructed skip lists.

### 31.2.2 skiptools.make\_skips\_with\_multiplied\_durations

skiptools.**make\_skips\_with\_multiplied\_durations** (*written\_duration*, *multiplied\_durations*)  
 Make *written\_duration* skips with *multiplied\_durations*:

```
>>> skiptools.make_skips_with_multiplied_durations(  
...     Duration(1, 4), [(1, 2), (1, 3), (1, 4), (1, 5)])  
Selection(Skip('s4 * 2'), Skip('s4 * 4/3'), Skip('s4 * 1'), Skip('s4 * 4/5'))
```

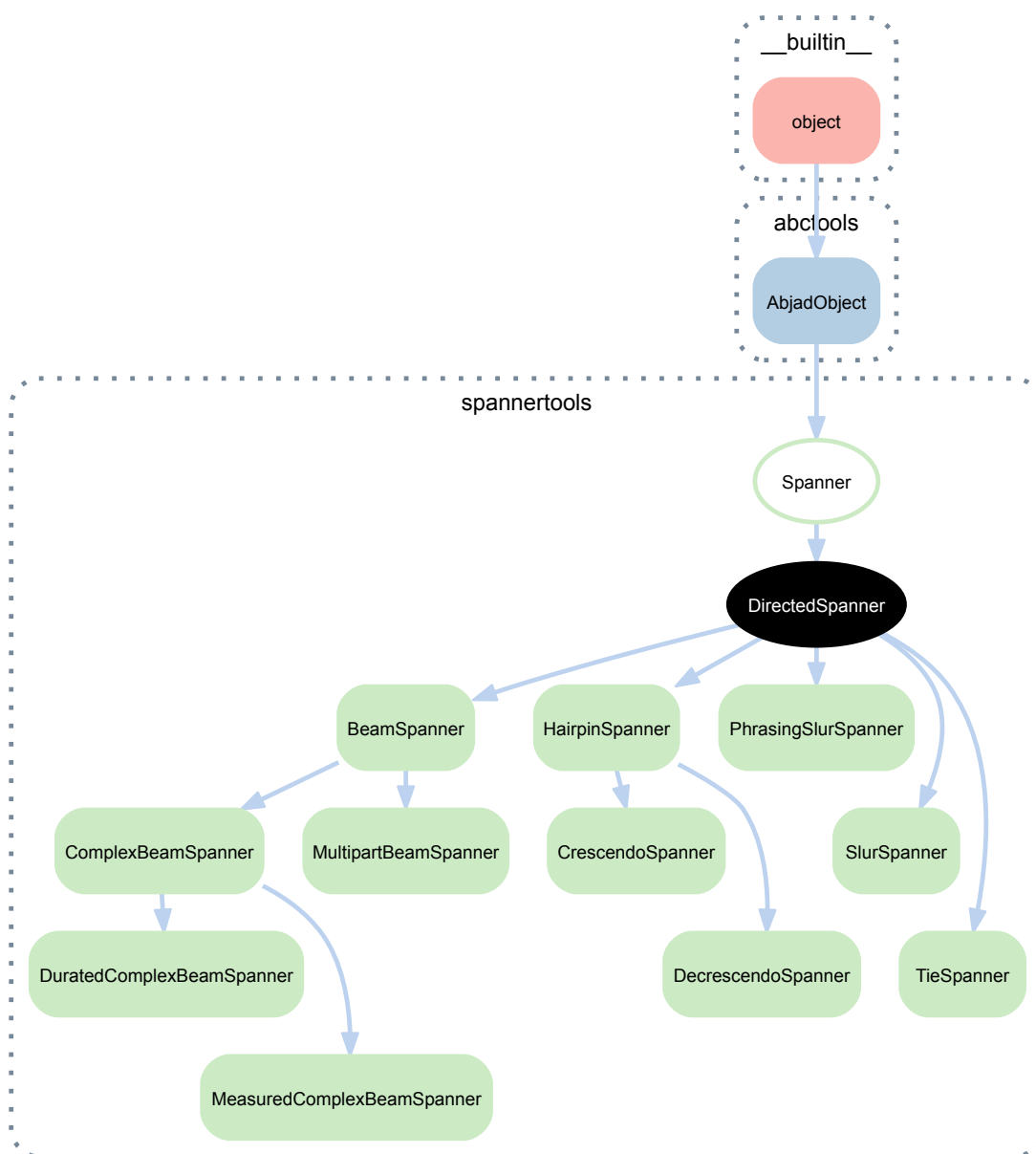
Useful for making invisible layout voices.

Returns list of skips.

## SPANNERTOOLS

### 32.1 Abstract classes

#### 32.1.1 spannertools.DirectedSpanner



**class** `spannertools.DirectedSpanner` (*components*=[], *direction*=None, *overrides*=None)  
Abstract base class for spanners which may take an “up” or “down” indication.

### Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

(`Spanner`) **.components**  
Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(`Spanner`) **.leaves**  
Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(`Spanner`) **.override**  
LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(`Spanner`) **.set**  
LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

### Read/write properties

`DirectedSpanner` **.direction**

### Methods

(`Spanner`) **.append** (*component*)  
Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1:])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .\_\_**call**\_\_ (*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .\_\_**contains**\_\_ (*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .\_\_**copy**\_\_ (\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .\_\_**eq**\_\_ (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .\_\_**getitem**\_\_ (*expr*)

Gets item from spanner.

Returns component.

(Spanner) .\_\_**len**\_\_ ()

Length of spanner.

Returns nonnegative integer.

(Spanner) .\_\_**lt**\_\_ (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .\_\_**ne**\_\_ (*expr*)

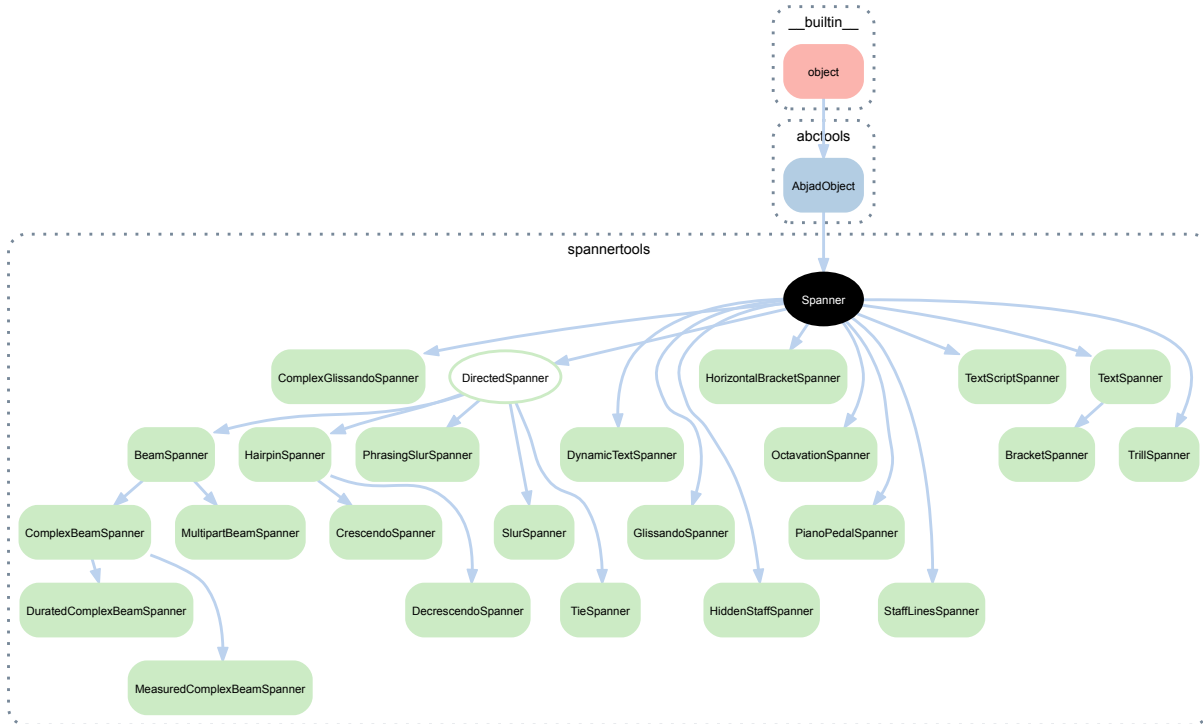
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.



(Spanner).**\_\_repr\_\_**()  
 Interpreter representation of spanner.  
 Returns string.

### 32.1.2 spannertools.Spanner



**class** spannertools.**Spanner** (*components=None, overrides=None*)

Any type of notation object that stretches horizontally and encompasses some number of notes, rest, chords, tuplets, measures, voices or other Abjad components.

Beams, slurs, hairpins, trills, glissandi and piano pedal brackets all stretch horizontally on the page to encompass multiple notes and all implement as Abjad spanners. That is, these spanner all have an obvious graphic reality with definite start-, stop- and midpoints.

Abjad also implements a number of spanners of a different type, such as tempo and instrument spanners, which mark a group of notes, rests, chords or measures as carrying a certain tempo or being played by a certain instrument.

The spanner class described here abstracts the functionality that all such spanners, both graphic and non-graphics, share. This shared functionality includes methods to add, remove, inspect and test components governed by the spanner, as well as basic formatting properties. The other spanner classes, such as beam and glissando, all inherit from this class and receive the functionality implemented here.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`Spanner.components`  
 Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

### **Spanner.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

### **Spanner.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

### **Spanner.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## **Methods**

### **Spanner.append(*component*)**

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

### **Spanner.append\_left(*component*)**

Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

**Spanner.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

**Spanner.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

**Spanner.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

**Spanner.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

`Spanner.fracture` (*i*, *direction=None*)

Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

`Spanner.fuse` (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

`Spanner.get_duration` (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

`Spanner.get_timespan` (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

`Spanner.index` (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

`Spanner.pop()`

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

`Spanner.pop_left()`

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

`Spanner.__call__(expr)`

Calls spanner on *expr*.

Same as `attach`.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

`Spanner.__contains__(expr)`

True when spanner contains *expr*. Otherwise false.

Returns boolean.

`Spanner.__copy__(*args)`

Copies spanner.

Does not copy spanner components.

Returns new spanner.

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`Spanner.__getitem__(expr)`

Gets item from spanner.

Returns component.

`Spanner.__len__()`

Length of spanner.

Returns nonnegative integer.

`Spanner.__lt__(expr)`

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

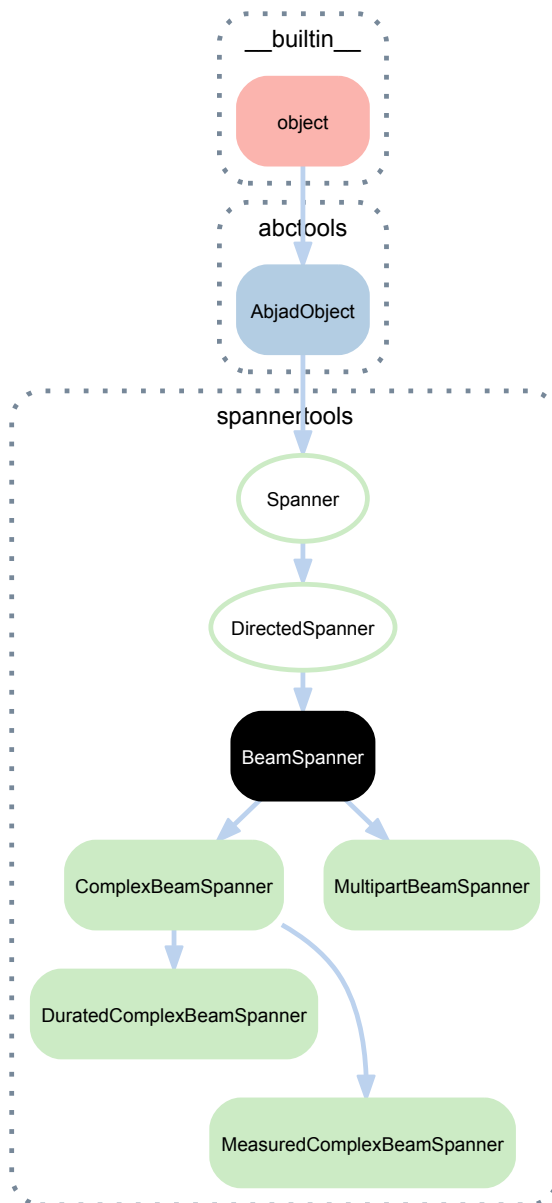
`Spanner.__repr__()`

Interpreter representation of spanner.

Returns string.

## 32.2 Concrete classes

### 32.2.1 spannertools.BeamSpanner



**class** `spannertools.BeamSpanner` (*components=None, direction=None, overrides=None*)  
 A beam spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'2")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner(staff[:4])
>>> show(staff)
```



## Bases

- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Spanner).components`

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).leaves`

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Spanner).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

`(DirectedSpanner).direction`

## Methods

`(Spanner).append(component)`

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```





```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) .**append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) .**attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) .**detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) .**extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Static methods

`BeamSpanner.is_beamable_component(expr)`

True when *expr* is a beamable component. Otherwise false.

```
>>> staff = Staff(r"r32 a'32 ( [ gs'32 fs''32 \staccato f''8 ) ]")
>>> staff.extend(r"r8 e''8 ( ef'2 )")
>>> show(staff)
```



(AbjadObject) .**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)  
 Gets item from spanner.  
 Returns component.

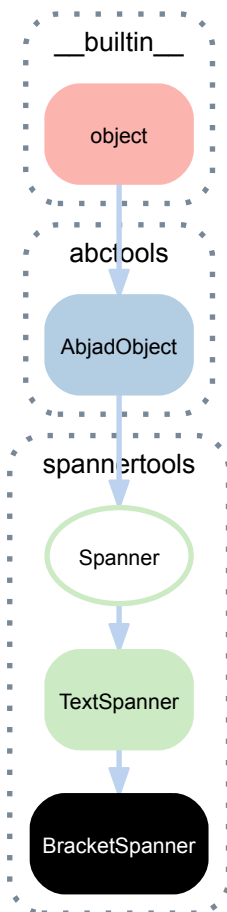
(Spanner) .**\_\_len\_\_**()  
 Length of spanner.  
 Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)  
 True when spanner is less than *expr*.  
 Trivial comparison to allow doctests to work.  
 Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Spanner) .**\_\_repr\_\_**()  
 Interpreter representation of spanner.  
 Returns string.

### 32.2.2 spannertools.BracketSpanner



**class** `spannertools.BracketSpanner` (*components=None, overrides=None*)  
A bracket spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.BracketSpanner(staff[:])  
BracketSpanner(c'8, d'8, e'8, f'8)
```

```
>>> show(staff)
```



Render 1.5-unit thick solid red spanner.

Draw nibs at beginning and end of spanner.

Do not draw nibs at line breaks.

Returns bracket spanner.

## Bases

- `spannertools.TextSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(`Spanner`) **.components**  
Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.BeamSpanner(voice[:2])  
>>> show(voice)
```



```
>>> spanner.components  
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(`Spanner`) **.leaves**  
Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.BeamSpanner(voice[:2])  
>>> show(voice)
```



```
>>> spanner.leaves  
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(`Spanner`) **.override**  
LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) . **set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Methods

(Spanner) . **append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) . **append\_left** (*component*)

Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) . **attach** (*components*)

Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) . **detach** ()

Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
 Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
 Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
 Fractures spanner at *direction* of component at index *i*.  
 Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
 Fuses spanner with contiguous *spanner*.



```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner).**\_\_call\_\_**(*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner).**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner).**\_\_copy\_\_**(\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner).**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner).**\_\_len\_\_**()

Length of spanner.

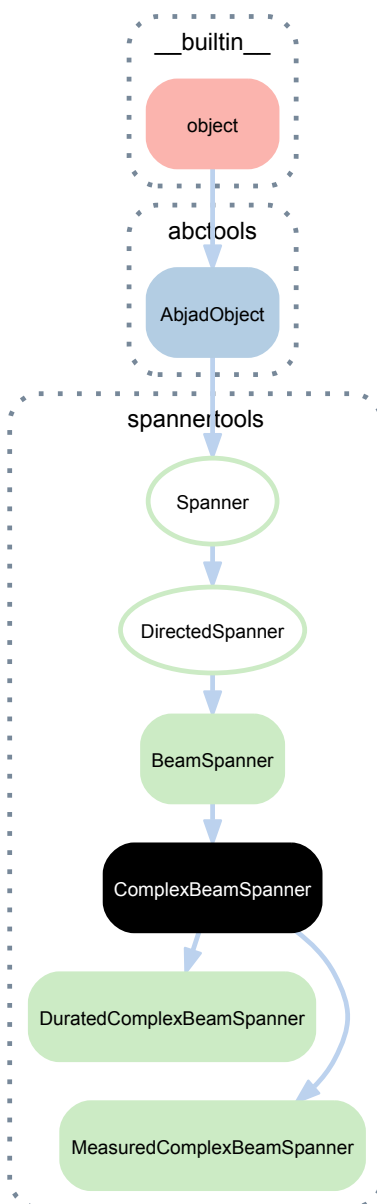
Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)  
 True when spanner is less than *expr*.  
 Trivial comparison to allow doctests to work.  
 Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Spanner) .**\_\_repr\_\_**()  
 Interpreter representation of spanner.  
 Returns string.

### 32.2.3 spannertools.ComplexBeamSpanner



**class** `spannertools.ComplexBeamSpanner` (*components=None*, *lone=False*, *direction=None*,  
*overrides=None*)  
 A complex beam spanner.

```
>>> staff = Staff("c'16 e'16 r16 f'16 g'2")
```

```
>>> show(staff)
```



```
>>> spannertools.ComplexBeamSpanner(staff[:4])
ComplexBeamSpanner(c'16, e'16, r16, f'16)
```

```
>>> show(staff)
```



Returns complex beam spanner.

## Bases

- `spannertools.BeamSpanner`
- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Spanner) **.components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) . **set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

(DirectedSpanner) . **direction**

ComplexBeamSpanner . **lone**

Beam lone leaf and force beam nibs to left:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='left')
```

Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='right')
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='both')
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone=True)
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone=False)
```

Set to 'left', 'right', 'both', true or false as shown above.

Ignore this setting when spanner contains more than one leaf.

## Methods

(Spanner) . **append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) . **fracture** (*i*, *direction=None*)

Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) . **fuse** (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) . **get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) .**get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) .**index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) .**pop** ()

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) .**pop\_left** ()

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Static methods

(BeamSpanner) .**is\_beamable\_component** (*expr*)

True when *expr* is a beamable component. Otherwise false.

```
>>> staff = Staff(r"r32 a'32 ( [ gs'32 fs''32 \staccato f''8 ) ]")
>>> staff.extend(r"r8 e''8 ( ef'2 )")
>>> show(staff)
```





```
>>> for leaf in staff.select_leaves():
...     beam = spannertools.BeamSpanner
...     result = beam.is_beamable_component(leaf)
...     print '{:<8}{}'.format(leaf, result)
...
r32      False
a'32     True
gs'32    True
fs''32   True
f''8     True
r8       False
e''8     True
ef'2     False
```

Returns boolean.

## Special methods

(Spanner) .**\_\_call\_\_**(*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_**(\*args)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

(*Spanner*) . **\_\_lt\_\_** (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(*AbjadObject*) . **\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

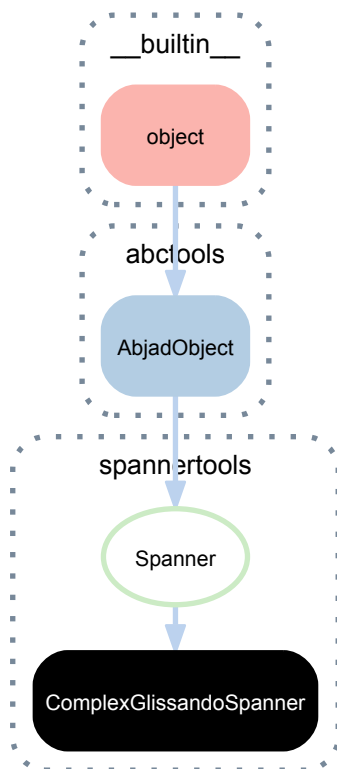
Returns boolean.

(*Spanner*) . **\_\_repr\_\_** ()

Interpreter representation of spanner.

Returns string.

### 32.2.4 spannertools.ComplexGlissandoSpanner



**class** spannertools.**ComplexGlissandoSpanner** (*components=None, overrides=None*)

A rest-skipping glissando spanner.

```
>>> staff = Staff("c'16 r r g' r8 c'8")
```

```
>>> spannertools.ComplexGlissandoSpanner(staff[:])
ComplexGlissandoSpanner(c'16, r16, r16, g'16, r8, c'8)
```

```
>>> show(staff)
```



Should be used with BeamSpanner for best effect, along with an override of stemlet-length, in order to generate stemlets over each invisible rest.

Format nonlast leaves in spanner with LilyPond glissando command.

Set all Rest instances to transparent.

Set all NoteColumns filled with silences to be skipped by glissandi.

Return *ComplexGlissandoSpanner* instance.

## Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Spanner) **.components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Methods

(Spanner) **.append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) .**append\_left** (*component*)

Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) .**attach** (*components*)

Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) .**detach** ()

Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) .**extend** (*components*)

Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .**\_\_call\_\_**(*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_**(\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

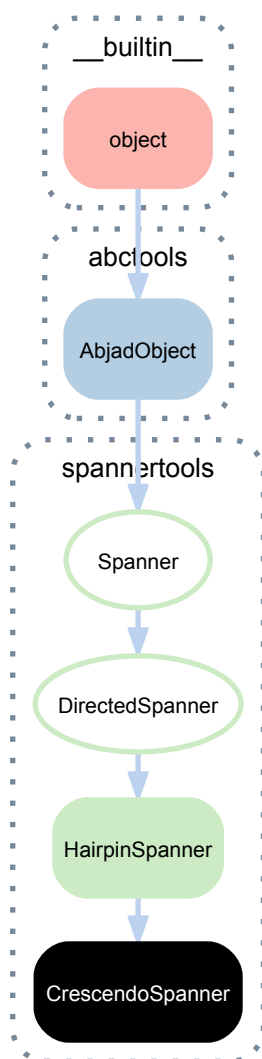
Returns boolean.

(Spanner) .**\_\_repr\_\_**()

Interpreter representation of spanner.

Returns string.

## 32.2.5 spannertools.CrescendoSpanner



**class** spannertools.**CrescendoSpanner** (*components=None, include\_rests=True, direction=None, overrides=None*)

A crescendo spanner that includes rests.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
```

```
>>> spannertools.CrescendoSpanner(staff[:], include_rests=True)
CrescendoSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> show(staff)
```



Abjad crescendo spanner that does not include rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
```

```
>>> spannertools.CrescendoSpanner(staff[:], include_rests=False)
CrescendoSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> show(staff)
```





Returns crescendo spanner.

## Bases

- `spannertools.HairpinSpanner`
- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Spanner) **.components**  
Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**  
Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**  
LilyPond grob override component plug-in.  
Returns LilyPond grob override component plug-in.

(Spanner) **.set**  
LilyPond context setting component plug-in.  
Returns LilyPond context setting component plug-in.

## Read/write properties

(DirectedSpanner) **.direction**

(HairpinSpanner) **.include\_rests**  
Get boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(
...     staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests
True
```

Set boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(
...     staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Set boolean.

**(HairpinSpanner).shape\_string**

Get hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string
'<'
```

Set hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string = '>'
>>> hairpin.shape_string
'>'
```

Set string.

**(HairpinSpanner).start\_dynamic\_string**

Get hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string
'p'
```

Set hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Set string.

**(HairpinSpanner).stop\_dynamic\_string**

Get hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string
'f'
```

Set hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Set string.

## Methods

(Spanner) .**append**(*component*)  
 Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) .**append\_left**(*component*)  
 Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) .**attach**(*components*)  
 Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) .**detach**()  
 Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) .**extend**(*components*)  
 Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner).**.get\_duration**(*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner).**.get\_timespan**(*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner).**.index**(*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner).**.pop**()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner).**.pop\_left**()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Static methods

(HairpinSpanner).**is\_hairpin\_shape\_string**(*arg*)

True when *arg* is a hairpin shape string. Otherwise false:

```
>>> spannertools.HairpinSpanner.is_hairpin_shape_string('<')
True
```

Returns boolean.

(HairpinSpanner).**is\_hairpin\_token**(*arg*)

True when *arg* is a hairpin token. Otherwise false:

```
>>> spannertools.HairpinSpanner.is_hairpin_token(('p', '<', 'f'))
True
```

```
>>> spannertools.HairpinSpanner.is_hairpin_token(('f', '<', 'p'))
False
```

Returns boolean.

## Special methods

(Spanner).**\_\_call\_\_**(*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner).**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner).**\_\_copy\_\_**(*\*args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)  
True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)  
Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()  
Length of spanner.

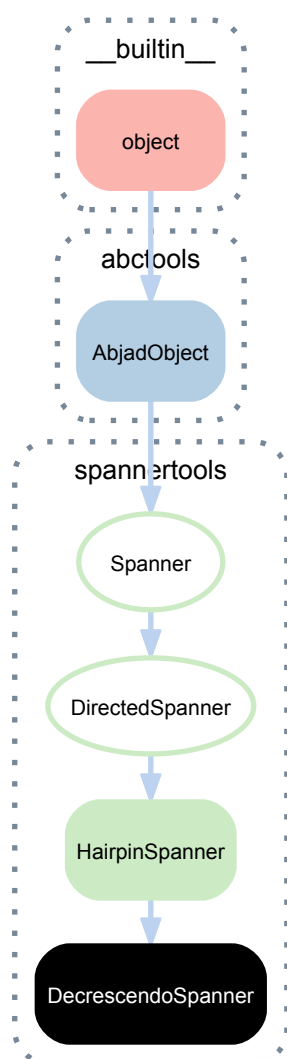
Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)  
True when spanner is less than *expr*.  
Trivial comparison to allow doctests to work.  
Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(Spanner) .**\_\_repr\_\_**()  
Interpreter representation of spanner.  
Returns string.

## 32.2.6 spannertools.DecrescendoSpanner



**class** spannertools.**DecrescendoSpanner** (*components=None, include\_rests=True, direction=None, overrides=None*)

A decrescendo spanner that includes rests.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
```

```
>>> spannertools.DecrescendoSpanner(staff[:], include_rests=True)
DecrescendoSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> show(staff)
```



Abjad decrescendo spanner that does not include rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
```

```
>>> spannertools.DecrescendoSpanner(staff[:], include_rests=False)
DecrescendoSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> show(staff)
```





Returns decrescendo spanner.

## Bases

- `spannertools.HairpinSpanner`
- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Spanner) **.components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

(DirectedSpanner) **.direction**

`(HairpinSpanner).include_rests`

Get boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(
...     staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests
True
```

Set boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(
...     staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Set boolean.

`(HairpinSpanner).shape_string`

Get hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string
'<'
```

Set hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string = '>'
>>> hairpin.shape_string
'>'
```

Set string.

`(HairpinSpanner).start_dynamic_string`

Get hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string
'p'
```

Set hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Set string.

`(HairpinSpanner).stop_dynamic_string`

Get hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string
'f'
```

Set hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Set string.

## Methods

(Spanner) .**append** (*component*)  
 Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) .**append\_left** (*component*)  
 Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) .**attach** (*components*)  
 Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) .**detach** ()  
 Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
 Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
 Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
 Fractures spanner at *direction* of component at index *i*.  
 Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
 Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Static methods

(HairpinSpanner).**is\_hairpin\_shape\_string**(*arg*)  
True when *arg* is a hairpin shape string. Otherwise false:

```
>>> spannertools.HairpinSpanner.is_hairpin_shape_string('<')
True
```

Returns boolean.

(HairpinSpanner).**is\_hairpin\_token**(*arg*)  
True when *arg* is a hairpin token. Otherwise false:

```
>>> spannertools.HairpinSpanner.is_hairpin_token(('p', '<', 'f'))
True
```

```
>>> spannertools.HairpinSpanner.is_hairpin_token(('f', '<', 'p'))
False
```

Returns boolean.

## Special methods

(Spanner).**\_\_call\_\_**(*expr*)  
Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner).**\_\_contains\_\_**(*expr*)  
True when spanner contains *expr*. Otherwise false.

Returns boolean.

(*Spanner*) .\_\_**copy**\_\_ (\**args*)  
Copies spanner.  
Does not copy spanner components.  
Returns new spanner.

(*AbjadObject*) .\_\_**eq**\_\_ (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(*Spanner*) .\_\_**getitem**\_\_ (*expr*)  
Gets item from spanner.  
Returns component.

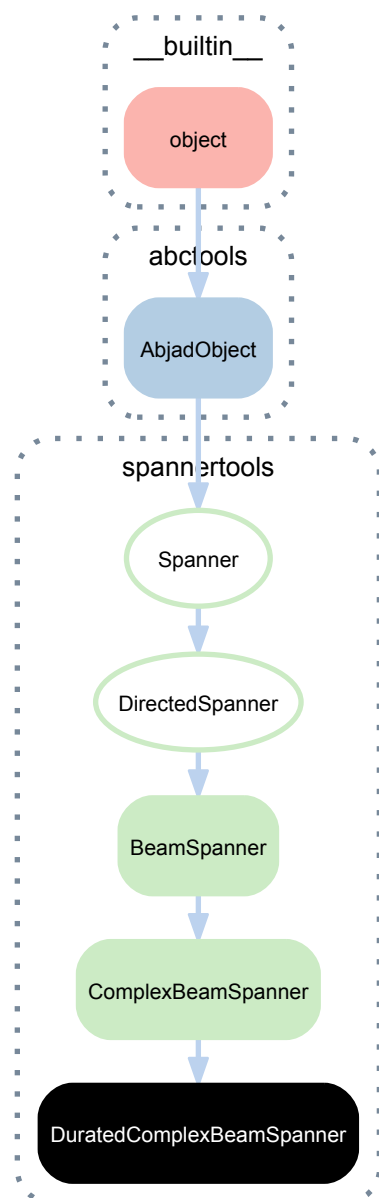
(*Spanner*) .\_\_**len**\_\_ ()  
Length of spanner.  
Returns nonnegative integer.

(*Spanner*) .\_\_**lt**\_\_ (*expr*)  
True when spanner is less than *expr*.  
Trivial comparison to allow doctests to work.  
Returns boolean.

(*AbjadObject*) .\_\_**ne**\_\_ (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(*Spanner*) .\_\_**repr**\_\_ ()  
Interpreter representation of spanner.  
Returns string.

### 32.2.7 spannertools.DuratedComplexBeamSpanner



**class** `spannertools.DuratedComplexBeamSpanner` (*components=None, durations=None, span=1, lone=False, direction=None, overrides=None*)

A durated complex beam spanner.

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
```

```
>>> show(staff)
```



```
>>> durations = [Duration(1, 8), Duration(1, 8)]
```

```
>>> beam = spannertools.DuratedComplexBeamSpanner(staff[:], durations, 1)
```

```
>>> show(staff)
```





Beam all beamable leaves in spanner explicitly.  
 Group leaves in spanner according to *durations*.  
 Span leaves between duration groups according to *span*.  
 Returns durated complex beam spanner.

## Bases

- `spannertools.ComplexBeamSpanner`
- `spannertools.BeamSpanner`
- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Spanner) **.components**  
 Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**  
 Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**  
 LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**  
 LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

(DirectedSpanner).**direction**

DuratedComplexBeamSpanner.**durations**

Get spanner leaf group durations:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = spannertools.DuratedComplexBeamSpanner(
...     staff[:], durations)
>>> beam.durations
[Duration(1, 8), Duration(1, 8)]
```

Set spanner leaf group durations:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = spannertools.DuratedComplexBeamSpanner(
...     staff[:], durations)
>>> beam.durations = [Duration(1, 4)]
>>> beam.durations
[Duration(1, 4)]
```

Set iterable.

(ComplexBeamSpanner).**lone**

Beam lone leaf and force beam nibs to left:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='left')
```

Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='right')
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='both')
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone=True)
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone=False)
```

Set to 'left', 'right', 'both', true or false as shown above.

Ignore this setting when spanner contains more than one leaf.

DuratedComplexBeamSpanner.**span**

Get top-level beam count:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = spannertools.DuratedComplexBeamSpanner(
...     staff[:], durations, 1)
>>> beam.span
1
```

Set top-level beam count:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = spannertools.DuratedComplexBeamSpanner(
...     staff[:], durations, 1)
>>> beam.span = 2
>>> beam.span
2
```

Set nonnegative integer.

## Methods

(Spanner) . **append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) . **append\_left** (*component*)

Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) . **attach** (*components*)

Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) . **detach** ()

Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)

Fuses *spanner* with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

`(Spanner).pop_left()`

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Static methods

`(BeamSpanner).is_beamable_component(expr)`

True when *expr* is a beamable component. Otherwise false.

```
>>> staff = Staff(r"r32 a'32 ( [ gs'32 fs''32 \staccato f''8 ) ]")
>>> staff.extend(r"r8 e''8 ( ef'2 )")
>>> show(staff)
```



```
>>> for leaf in staff.select_leaves():
...     beam = spannertools.BeamSpanner
...     result = beam.is_beamable_component(leaf)
...     print '{:<8}{}'.format(leaf, result)
...
r32      False
a'32     True
gs'32    True
fs''32   True
f''8     True
r8       False
e''8     True
ef'2     False
```

Returns boolean.

## Special methods

`(Spanner).__call__(expr)`

Calls spanner on *expr*.

Same as `attach`.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_**(\*args)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

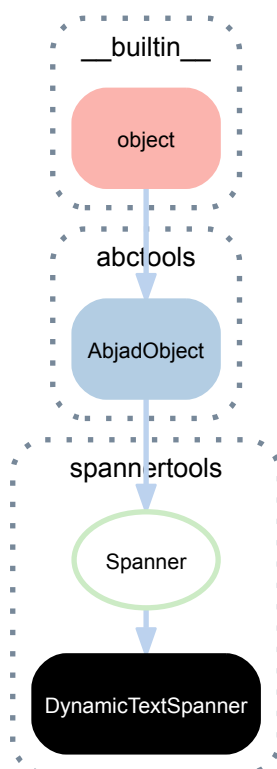
Returns boolean.

(Spanner) .**\_\_repr\_\_**()

Interpreter representation of spanner.

Returns string.

### 32.2.8 spannertools.DynamicTextSpanner



**class** `spannertools.DynamicTextSpanner` (*components=None, mark='', overrides=None*)  
 A dynamic text spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.DynamicTextSpanner(staff[:], 'f')
DynamicTextSpanner(c'8, d'8, e'8, f'8)
```

```
>>> show(staff)
```



Format dynamic *mark* at first leaf in spanner.

Returns dynamic text spanner.

#### Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

(`Spanner`) . **components**  
 Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```





```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

DynamicTextSpanner **.mark**

Get dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> dynamic_text_spanner = \
...     spannertools.DynamicTextSpanner(staff[:], 'f')
>>> dynamic_text_spanner.mark
'f'
```

Set dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> dynamic_text_spanner = \
...     spannertools.DynamicTextSpanner(staff[:], 'f')
>>> dynamic_text_spanner.mark = 'p'
>>> dynamic_text_spanner.mark
'p'
```

Set string.

## Methods

(Spanner) **.append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) .**append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) .**attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) .**detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) .**extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
 Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
 Fractures spanner at *direction* of component at index *i*.  
 Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
 Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .**\_\_call\_\_**(*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_**(\*args)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

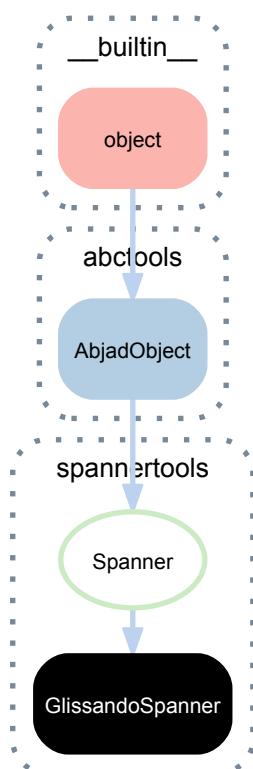
Returns boolean.

(Spanner) .**\_\_repr\_\_**()

Interpreter representation of spanner.

Returns string.

### 32.2.9 spannertools.GlissandoSpanner



**class** `spannertools.GlissandoSpanner` (*components=None, overrides=None*)  
 A glissando spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.GlissandoSpanner(staff[:])
GlissandoSpanner(c'8, d'8, e'8, f'8)
```

```
>>> show(staff)
```



Format nonlast leaves in spanner with LilyPond glissando command.

Returns glissando spanner.

#### Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

(`Spanner`) . **components**  
 Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Methods

(Spanner) **.append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)

Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.



Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) .**fuse**(*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) .**get\_duration**(*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) .**get\_timespan**(*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) .**index**(*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])  
0
```

Returns nonnegative integer.

(Spanner) **.pop()**

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.SlurSpanner(voice[:])  
>>> show(voice)
```



```
>>> note = spanner.pop()  
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left()**

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.SlurSpanner(voice[:])  
>>> show(voice)
```



```
>>> note = spanner.pop_left()  
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) **.\_\_call\_\_(expr)**

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")  
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()  
>>> beam = beam(staff[:])  
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(*Spanner*) .\_\_contains\_\_(*expr*)  
True when spanner contains *expr*. Otherwise false.  
Returns boolean.

(*Spanner*) .\_\_copy\_\_(\**args*)  
Copies spanner.  
Does not copy spanner components.  
Returns new spanner.

(*AbjadObject*) .\_\_eq\_\_(*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

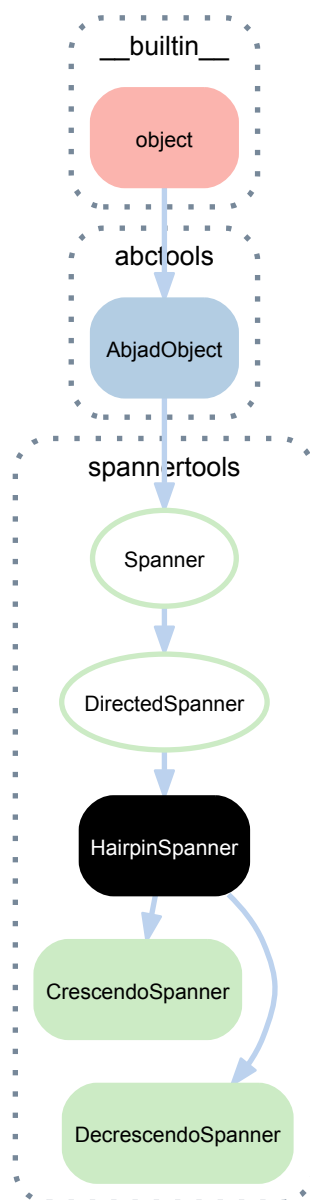
(*Spanner*) .\_\_getitem\_\_(*expr*)  
Gets item from spanner.  
Returns component.

(*Spanner*) .\_\_len\_\_()  
Length of spanner.  
Returns nonnegative integer.

(*Spanner*) .\_\_lt\_\_(*expr*)  
True when spanner is less than *expr*.  
Trivial comparison to allow doctests to work.  
Returns boolean.

(*AbjadObject*) .\_\_ne\_\_(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(*Spanner*) .\_\_repr\_\_()  
Interpreter representation of spanner.  
Returns string.

32.2.10 `spannertools.HairpinSpanner`

**class** `spannertools.HairpinSpanner` (*components=None, descriptor='<', include\_rests=True, direction=None, overrides=None*)

A dynamic hairpin spanner.

**Example 1.** Hairpin spanner that does not include rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> spannertools.HairpinSpanner(
...     staff[:], 'p < f', include_rests=False)
HairpinSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> show(staff)
```



**Example 2.** Hairpin spanner that includes rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> spannertools.HairpinSpanner(
```

```
...      staff[:], 'p < f', include_rests=True)
HairpinSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> show(staff)
```



Returns hairpin spanner.

## Bases

- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Spanner).components`

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).leaves`

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Spanner).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

(DirectedSpanner).**direction**

HairpinSpanner.**include\_rests**

Get boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(
...     staff[:, 'p < f', include_rests=True)
>>> hairpin.include_rests
True
```

Set boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(
...     staff[:, 'p < f', include_rests=True)
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Set boolean.

HairpinSpanner.**shape\_string**

Get hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:, 'p < f')
>>> hairpin.shape_string
'<'
```

Set hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:, 'p < f')
>>> hairpin.shape_string = '>'
>>> hairpin.shape_string
'>'
```

Set string.

HairpinSpanner.**start\_dynamic\_string**

Get hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:, 'p < f')
>>> hairpin.start_dynamic_string
'p'
```

Set hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:, 'p < f')
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Set string.

HairpinSpanner.**stop\_dynamic\_string**

Get hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:, 'p < f')
>>> hairpin.stop_dynamic_string
'f'
```

Set hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Set string.

## Methods

(Spanner) **.append** (*component*)  
 Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
 Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
 Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
 Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.



(Spanner) .**fuse**(*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) .**get\_duration**(*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) .**get\_timespan**(*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) .**index**(*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) .**pop**()

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left()**

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Static methods

HairpinSpanner.**is\_hairpin\_shape\_string**(arg)

True when arg is a hairpin shape string. Otherwise false:

```
>>> spannertools.HairpinSpanner.is_hairpin_shape_string('<')
True
```

Returns boolean.

HairpinSpanner.**is\_hairpin\_token**(arg)

True when arg is a hairpin token. Otherwise false:

```
>>> spannertools.HairpinSpanner.is_hairpin_token(('p', '<', 'f'))
True
```

```
>>> spannertools.HairpinSpanner.is_hairpin_token(('f', '<', 'p'))
False
```

Returns boolean.

## Special methods

(Spanner) **.\_\_call\_\_**(expr)

Calls spanner on expr.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(*Spanner*) .\_\_contains\_\_ (*expr*)  
True when spanner contains *expr*. Otherwise false.  
Returns boolean.

(*Spanner*) .\_\_copy\_\_ (\**args*)  
Copies spanner.  
Does not copy spanner components.  
Returns new spanner.

(*AbjadObject*) .\_\_eq\_\_ (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(*Spanner*) .\_\_getitem\_\_ (*expr*)  
Gets item from spanner.  
Returns component.

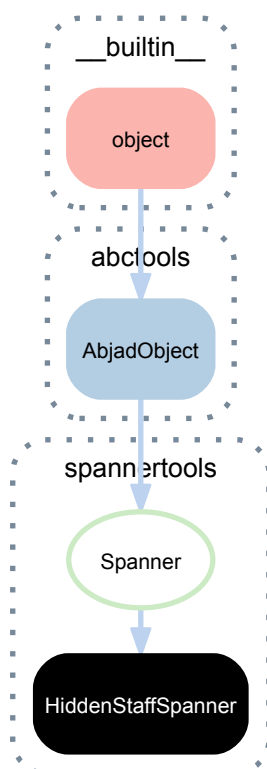
(*Spanner*) .\_\_len\_\_ ()  
Length of spanner.  
Returns nonnegative integer.

(*Spanner*) .\_\_lt\_\_ (*expr*)  
True when spanner is less than *expr*.  
Trivial comparison to allow doctests to work.  
Returns boolean.

(*AbjadObject*) .\_\_ne\_\_ (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(*Spanner*) .\_\_repr\_\_ ()  
Interpreter representation of spanner.  
Returns string.

### 32.2.11 spannertools.HiddenStaffSpanner



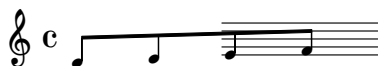
**class** `spannertools.HiddenStaffSpanner` (*components=None, overrides=None*)

A hidden staff spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.HiddenStaffSpanner(staff[:2])
HiddenStaffSpanner(c'8, d'8)
```

```
>>> show(staff)
```



Hide staff behind leaves in spanner.

Returns hidden staff spanner.

#### Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

(`Spanner`) . **components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Methods

(Spanner) **.append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)

Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) .**fuse** (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) .**get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) .**get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) .**index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop()**

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left()**

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) **.\_\_call\_\_(expr)**

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.



(*Spanner*) .\_\_contains\_\_ (*expr*)  
True when spanner contains *expr*. Otherwise false.  
Returns boolean.

(*Spanner*) .\_\_copy\_\_ (\**args*)  
Copies spanner.  
Does not copy spanner components.  
Returns new spanner.

(*AbjadObject*) .\_\_eq\_\_ (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

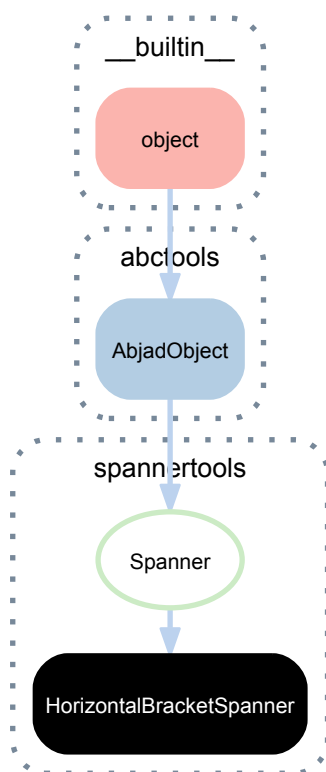
(*Spanner*) .\_\_getitem\_\_ (*expr*)  
Gets item from spanner.  
Returns component.

(*Spanner*) .\_\_len\_\_ ()  
Length of spanner.  
Returns nonnegative integer.

(*Spanner*) .\_\_lt\_\_ (*expr*)  
True when spanner is less than *expr*.  
Trivial comparison to allow doctests to work.  
Returns boolean.

(*AbjadObject*) .\_\_ne\_\_ (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(*Spanner*) .\_\_repr\_\_ ()  
Interpreter representation of spanner.  
Returns string.

32.2.12 `spannertools.HorizontalBracketSpanner`

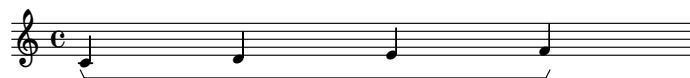
**class** `spannertools.HorizontalBracketSpanner` (*components=None, overrides=None*)  
 A horizontal bracket spanner.

```
>>> voice = Voice("c'4 d'4 e'4 f'4")
>>> voice.engraver_consists.append('Horizontal_bracket_engraver')
```

```
>>> horizontal_bracket_spanner = \
...     spannertools.HorizontalBracketSpanner(voice[:])
```

```
>>> horizontal_bracket_spanner
HorizontalBracketSpanner(c'4, d'4, e'4, f'4)
```

```
>>> show(voice)
```



Returns horizontal bracket spanner.

**Bases**

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

**Read-only properties**

(`Spanner`) . **components**  
 Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**  
Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**  
LilyPond grob override component plug-in.  
Returns LilyPond grob override component plug-in.

(Spanner) **.set**  
LilyPond context setting component plug-in.  
Returns LilyPond context setting component plug-in.

## Methods

(Spanner) **.append** (*component*)  
Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) .**fracture** (*i*, *direction=None*)

Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) .**fuse** (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) .**get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) .**get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) .**index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])  
0
```

Returns nonnegative integer.

(Spanner) **.pop()**

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.SlurSpanner(voice[:])  
>>> show(voice)
```



```
>>> note = spanner.pop()  
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left()**

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.SlurSpanner(voice[:])  
>>> show(voice)
```



```
>>> note = spanner.pop_left()  
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) **.\_\_call\_\_()** (*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")  
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()  
>>> beam = beam(staff[:])  
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_** (*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_** (\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_** (*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_** ()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_** (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_** (*expr*)

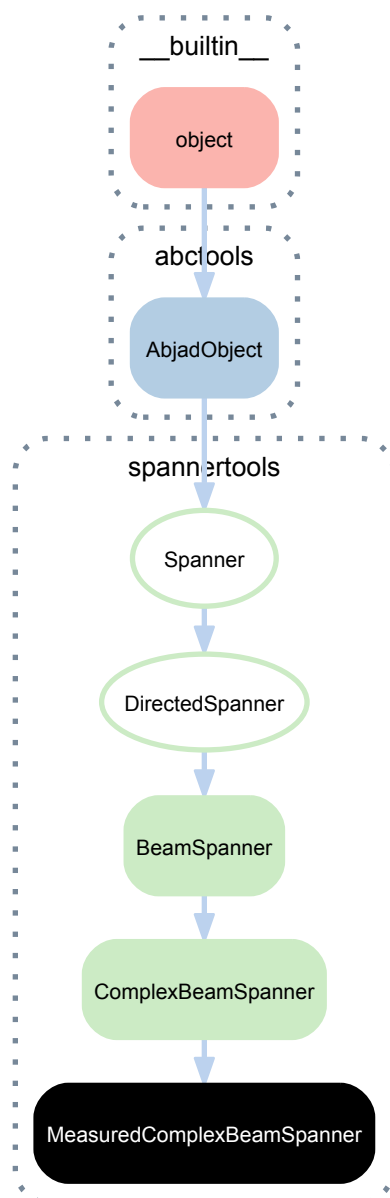
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_repr\_\_** ()

Interpreter representation of spanner.

Returns string.

32.2.13 `spannertools.MeasuredComplexBeamSpanner`

**class** `spannertools.MeasuredComplexBeamSpanner` (*components=None, lone=False, span=1, direction=None, overrides=None*)

A measured complex beam spanner.

```
>>> staff = Staff(
...     [Measure((2, 16), "c'16 d'16"),
...     Measure((2, 16), "e'16 f'16")])
```

```
>>> show(staff)
```



```
>>> spannertools.MeasuredComplexBeamSpanner(staff.select_leaves())
MeasuredComplexBeamSpanner(c'16, d'16, e'16, f'16)
```

```
>>> show(staff)
```





Beam leaves in spanner explicitly.

Group leaves by measures.

Format top-level *span* beam between measures.

Returns measured complex beam spanner.

## Bases

- `spannertools.ComplexBeamSpanner`
- `spannertools.BeamSpanner`
- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Spanner) **.components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

(DirectedSpanner) .**direction**

(ComplexBeamSpanner) .**lone**

Beam lone leaf and force beam nibs to left:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='left')
```

Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='right')
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone='both')
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone=True)
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeamSpanner([note], lone=False)
```

Set to 'left', 'right', 'both', true or false as shown above.

Ignore this setting when spanner contains more than one leaf.

MeasuredComplexBeamSpanner .**span**

Get top-level beam count:

```
>>> staff = Staff([
...     Measure((2, 16), "c'16 d'16"),
...     Measure((2, 16), "e'16 f'16")])
>>> beam = spannertools.MeasuredComplexBeamSpanner(staff.select_leaves())
>>> beam.span
1
```

Set top-level beam count:

```
>>> staff = Staff([
...     Measure((2, 16), "c'16 d'16"),
...     Measure((2, 16), "e'16 f'16")])
>>> beam = spannertools.MeasuredComplexBeamSpanner(staff.select_leaves())
>>> beam.span = 2
>>> beam.span
2
```

Set nonnegative integer.

## Methods

(Spanner) .**append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
 Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
 Fractures spanner at *direction* of component at index *i*.  
 Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
 Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Static methods

(BeamSpanner) **.is\_beamable\_component** (*expr*)  
True when *expr* is a beamable component. Otherwise false.

```
>>> staff = Staff(r"r32 a'32 ( [ gs'32 fs''32 \staccato f''8 ) ]")
>>> staff.extend(r"r8 e''8 ( ef'2 )")
>>> show(staff)
```



```
>>> for leaf in staff.select_leaves():
...     beam = spannertools.BeamSpanner
...     result = beam.is_beamable_component(leaf)
...     print '{:<8}{}'.format(leaf, result)
...
r32      False
a'32     True
gs'32    True
fs''32   True
f''8     True
r8       False
e''8     True
ef'2     False
```

Returns boolean.

## Special methods

(Spanner) **.\_\_call\_\_** (*expr*)  
Calls spanner on *expr*.  
Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) **.\_\_contains\_\_** (*expr*)  
True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) **.\_\_copy\_\_** (*\*args*)  
Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_** (*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_** ()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_** (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_** (*expr*)

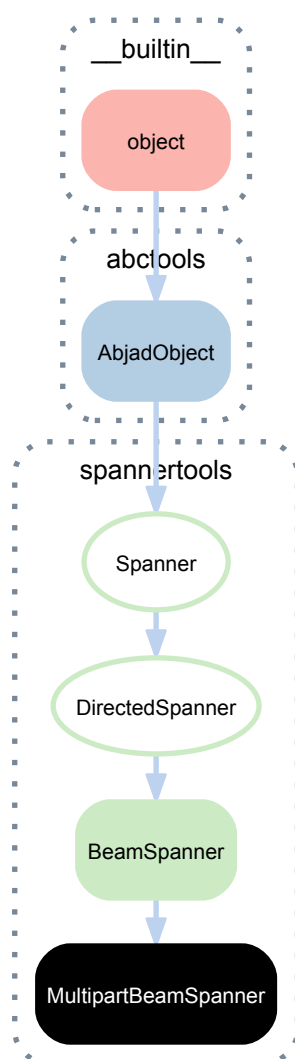
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_repr\_\_** ()

Interpreter representation of spanner.

Returns string.

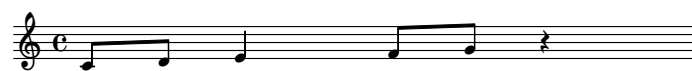
32.2.14 `spannertools.MultipartBeamSpanner`

**class** `spannertools.MultipartBeamSpanner` (*components=None*, *direction=None*, *overrides=None*)

A multipart beam spanner.

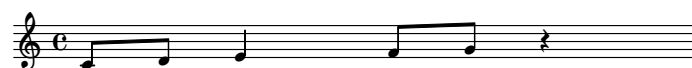
```
>>> staff = Staff("c'8 d'8 e'4 f'8 g'8 r4")
```

```
>>> show(staff)
```



```
>>> spannertools.MultipartBeamSpanner(staff[:])
MultipartBeamSpanner(c'8, d'8, e'4, f'8, g'8, r4)
```

```
>>> show(staff)
```



Avoid rests.

Avoid large-duration notes.

Returns multipart beam spanner.



## Bases

- `spannertools.BeamSpanner`
- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Spanner).components`

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).leaves`

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Spanner).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

`(DirectedSpanner).direction`

## Methods

`(Spanner).append(component)`

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) .**extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) .**fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) .**fuse** (*spanner*)  
Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:  
...     x  
BeamSpanner(c'8, d'8)  
BeamSpanner(e'8, f'8)  
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.BeamSpanner(voice[2:])  
>>> show(voice)
```



```
>>> spanner.index(voice[-2])  
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.SlurSpanner(voice[:])  
>>> show(voice)
```



```
>>> note = spanner.pop()  
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")  
>>> spanner = spannertools.SlurSpanner(voice[:])  
>>> show(voice)
```



```
>>> note = spanner.pop_left()  
>>> show(voice)
```



Returns component.

## Static methods

(BeamSpanner) **.is\_beamable\_component** (*expr*)  
True when *expr* is a beamable component. Otherwise false.

```
>>> staff = Staff(r"r32 a'32 ( [ gs'32 fs''32 \staccato f''8 ] )")
>>> staff.extend(r"r8 e''8 ( ef'2 )")
>>> show(staff)
```



```
>>> for leaf in staff.select_leaves():
...     beam = spannertools.BeamSpanner
...     result = beam.is_beamable_component(leaf)
...     print '{:<8}{}'.format(leaf, result)
...
r32      False
a'32     True
gs'32    True
fs''32   True
f''8     True
r8       False
e''8     True
ef'2     False
```

Returns boolean.

## Special methods

(Spanner) **.\_\_call\_\_** (*expr*)  
Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) **.\_\_contains\_\_** (*expr*)  
True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) **.\_\_copy\_\_** (\*args)  
Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_** (*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_** ()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_** (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

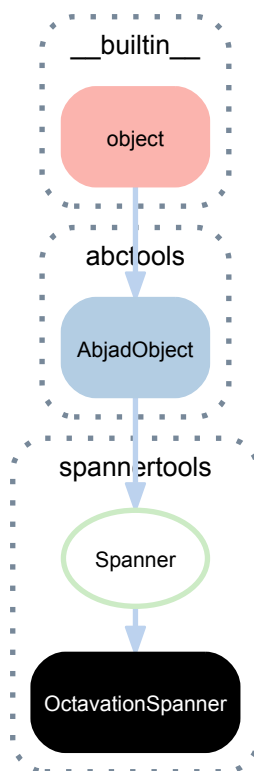
Returns boolean.

(Spanner) .**\_\_repr\_\_** ()

Interpreter representation of spanner.

Returns string.

### 32.2.15 spannertools.OctavationSpanner



**class** `spannertools.OctavationSpanner` (*components=None, start=1, stop=0, overrides=None*)  
 An octavation spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> spanner = spannertools.OctavationSpanner(staff[:], start=1)
>>> show(staff)
```



## Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Spanner).components`  
 Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).leaves`  
 Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).override`  
 LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Spanner).set`  
 LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

`OctavationSpanner.start`

Get octavation start:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> octavation = spannertools.OctavationSpanner(
...     staff[:], start=1)
>>> octavation.start
1
```

Set octavation start:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> octavation = spannertools.OctavationSpanner(
...     staff[:], start=1)
>>> octavation.start
1
```

Set integer.

`OctavationSpanner.stop`

Get octavation stop:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> octavation = spannertools.OctavationSpanner(
...     staff[:], start=2, stop=1)
>>> octavation.stop
1
```

Set octavation stop:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> octavation = spannertools.OctavationSpanner(
...     staff[:], start=2, stop=1)
>>> octavation.stop = 0
>>> octavation.stop
0
```

Set integer.

## Methods

`OctavationSpanner.adjust_automatically` (*ottava\_breakpoint=None*, *quinde-*  
*cisima\_breakpoint=None*)

Adjusts octavation spanner start and stop automatically according to *ottava\_breakpoint* and *quinde-*  
*cisima\_breakpoint*.

```
>>> measure = Measure((4, 8), "c'''8 d'''8 ef'''8 f'''8")
>>> octavation = spannertools.OctavationSpanner(measure[:])
>>> show(measure)
```



```
>>> octavation.adjust_automatically(ottava_breakpoint=14)
>>> show(measure)
```



Adjusts start and stop according to the diatonic pitch number of the maximum pitch in spanner.

Returns none.

(Spanner) . **append** (*component*)

Appends *component* to spanner.



```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
 Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
 Fractures spanner at *direction* of component at index *i*.  
 Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
 Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .\_\_**call**\_\_ (*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .\_\_**contains**\_\_ (*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .\_\_**copy**\_\_ (\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .\_\_**eq**\_\_ (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .\_\_**getitem**\_\_ (*expr*)

Gets item from spanner.

Returns component.

(Spanner) .\_\_**len**\_\_ ()

Length of spanner.

Returns nonnegative integer.

(Spanner) .\_\_**lt**\_\_ (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

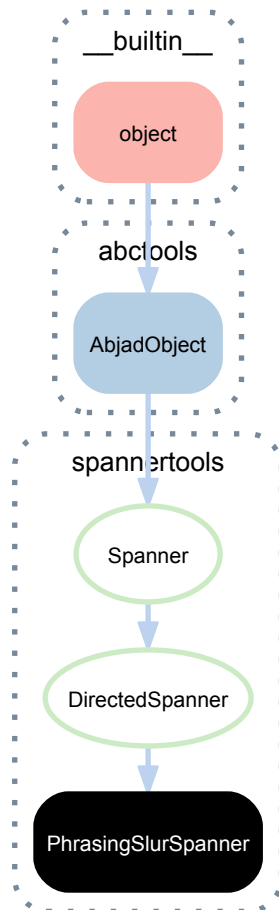
(AbjadObject) .\_\_**ne**\_\_ (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(Spanner).__repr__()`  
 Interpreter representation of spanner.  
 Returns string.

### 32.2.16 spannertools.PhrasingSlurSpanner



**class** `spannertools.PhrasingSlurSpanner` (*components=None, direction=None, overrides=None*)  
 A phrasing slur spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.PhrasingSlurSpanner(staff[:])
PhrasingSlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> show(staff)
```



Returns phrasing slur spanner.

#### Bases

- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`

- `__builtin__.object`

## Read-only properties

(Spanner) **.components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

(DirectedSpanner) **.direction**

## Methods

(Spanner) **.append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
 Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
 Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
 Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
 Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
 Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) . **fracture** (*i*, *direction=None*)

Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) . **fuse** (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.



(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) **.\_\_call\_\_** (*expr*)  
Calls spanner on *expr*.

Same as `attach`.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_**(\*args)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

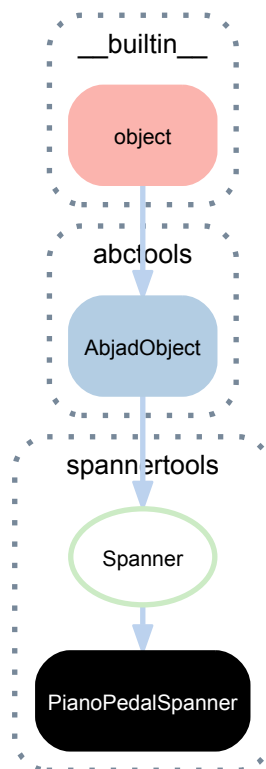
Returns boolean.

(Spanner) .**\_\_repr\_\_**()

Interpreter representation of spanner.

Returns string.

### 32.2.17 spannertools.PianoPedalSpanner



**class** spannertools.**PianoPedalSpanner** (*components=None, overrides=None*)

A piano pedal spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.PianoPedalSpanner(staff[:])
PianoPedalSpanner(c'8, d'8, e'8, f'8)
```

```
>>> show(staff)
```



Returns piano pedal spanner.

#### Bases

- spannertools.Spanner
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

(Spanner).**components**  
Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

PianoPedalSpanner **.kind**

Get piano pedal spanner kind:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(staff[:])
>>> spanner.kind
'sustain'
```

Set piano pedal spanner kind:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(staff[:])
>>> spanner.kind = 'sostenuto'
>>> spanner.kind
'sostenuto'
```

Acceptable values 'sustain', 'sostenuto', 'corda'.

PianoPedalSpanner **.style**

Get piano pedal spanner style:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(staff[:])
>>> spanner.style
'mixed'
```

Set piano pedal spanner style:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(staff[:])
>>> spanner.style = 'bracket'
>>> spanner.style
'bracket'
```

Acceptable values 'mixed', 'bracket', 'text'.

## Methods

(Spanner) .**append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) .**append\_left** (*component*)

Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) .**attach** (*components*)

Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) .**detach** ()

Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

### Special methods

(Spanner).**\_\_call\_\_**(*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner).**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner).**\_\_copy\_\_**(\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner).**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner).**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

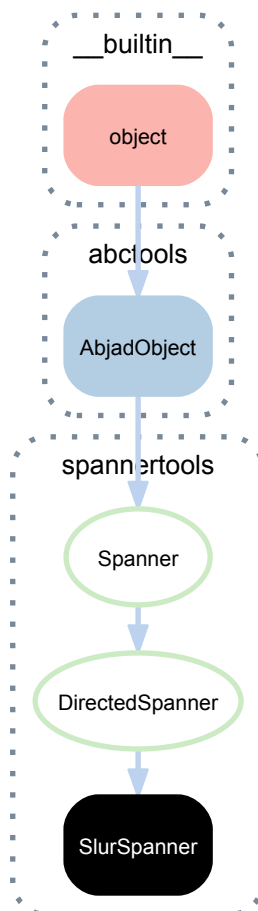


(*Spanner*) . **\_\_lt\_\_** (*expr*)  
 True when *spanner* is less than *expr*.  
 Trivial comparison to allow doctests to work.  
 Returns boolean.

(*AbjadObject*) . **\_\_ne\_\_** (*expr*)  
 True when ID of *expr* does not equal ID of *Abjad* object.  
 Returns boolean.

(*Spanner*) . **\_\_repr\_\_** ()  
 Interpreter representation of *spanner*.  
 Returns string.

### 32.2.18 spannertools.SlurSpanner



**class** spannertools.**SlurSpanner** (*components=None, direction=None, overrides=None*)  
 A slur spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.SlurSpanner(staff[:])
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> show(staff)
```



Returns slur spanner.

## Bases

- `spannertools.DirectedSpanner`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Spanner).components`

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).leaves`

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

`(Spanner).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Spanner).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

`(DirectedSpanner).direction`

## Methods

`(Spanner).append(component)`

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) **.fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) **.fuse** (*spanner*)  
Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .\_\_**call**\_\_ (*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .\_\_**contains**\_\_ (*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .\_\_**copy**\_\_ (\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .\_\_**eq**\_\_ (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .\_\_**getitem**\_\_ (*expr*)

Gets item from spanner.

Returns component.

(Spanner) .\_\_**len**\_\_ ()

Length of spanner.

Returns nonnegative integer.

(Spanner) .\_\_**lt**\_\_ (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

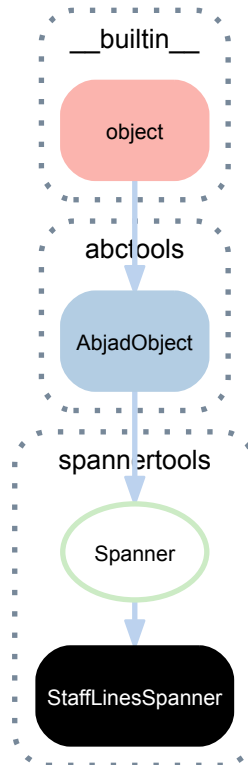
(AbjadObject) .\_\_**ne**\_\_ (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(Spanner).__repr__()`  
 Interpreter representation of spanner.  
 Returns string.

### 32.2.19 spannertools.StaffLinesSpanner



**class** `spannertools.StaffLinesSpanner` (*components=None, lines=5, overrides=None*)  
 A staff lines spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.StaffLinesSpanner(staff[:2], 1)
StaffLinesSpanner(c'8, d'8)
```

```
>>> show(staff)
```



Staff lines spanner handles changing either the line-count or the line-positions property of the `StaffSymbol` grob, as well as automatically stopping and restarting the staff so that the change may take place.

Returns staff lines spanner.

#### Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Spanner) . **components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) . **leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) . **override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) . **set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

StaffLinesSpanner . **lines**

Get staff lines spanner line count:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.StaffLinesSpanner(staff[:2], 1)
>>> spanner.lines
1
```

Set staff lines spanner line count:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.StaffLinesSpanner(staff[:2], 1)
>>> spanner.lines = 2
>>> spanner.lines
2
```

Set integer.

## Methods

(Spanner) . **append** (*component*)

Appends *component* to spanner.



```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) .**extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) .**fracture** (*i*, *direction=None*)  
Fractures spanner at *direction* of component at index *i*.  
Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) .**fuse** (*spanner*)  
Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```





```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) **.get\_duration** (*in\_seconds=False*)  
Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)  
Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)  
Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()  
Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()  
Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .**\_\_call\_\_**(*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_**(\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

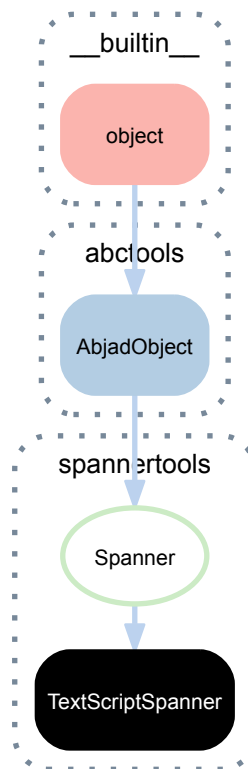
(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Spanner).**\_\_repr\_\_**()  
 Interpreter representation of spanner.  
 Returns string.

### 32.2.20 spannertools.TextScriptSpanner



**class** spannertools.**TextScriptSpanner** (components=None, overrides=None)  
 A text script spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spanner = spannertools.TextScriptSpanner(staff[:])
>>> spanner.override.text_script.color = 'red'
>>> markuptools.Markup(r'\italic { espressivo }', Up)(staff[1])
Markup(MarkupCommand('italic', ['espressivo']), direction=Up)(d'8)
```

```
>>> show(staff)
```



Override LilyPond TextScript grob.

Returns text script spanner.

#### Bases

- spannertools.Spanner
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

(Spanner) .**components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) .**leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) .**override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) .**set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Methods

(Spanner) .**append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) .**append\_left** (*component*)

Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) .**fracture** (*i*, *direction=None*)

Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) .**fuse** (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) .**get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) .**get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.



Returns timespan.

(Spanner) .**index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) .**pop** ()

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) .**pop\_left** ()

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .**\_\_call\_\_** (*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_**(\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

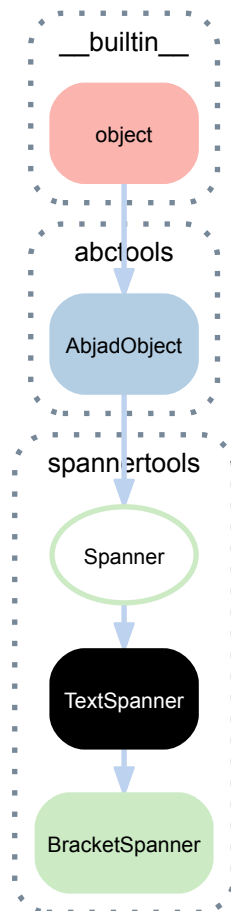
Returns boolean.

(Spanner) .**\_\_repr\_\_**()

Interpreter representation of spanner.

Returns string.

### 32.2.21 spannertools.TextSpanner



**class** `spannertools.TextSpanner` (*components=None, overrides=None*)  
 A text spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> text_spanner = spannertools.TextSpanner()
>>> grob = text_spanner.override.text_spanner
>>> markup_command = markuptools.MarkupCommand('italic', 'foo')
>>> markup_command = markuptools.MarkupCommand('bold', markup_command)
>>> left_markup = markuptools.Markup(markup_command)
>>> grob.bound_details__left__text = left_markup
>>> pair = schemetools.SchemePair(0, -1)
>>> markup_command = markuptools.MarkupCommand('draw-line', pair)
>>> right_markup = markuptools.Markup(markup_command)
>>> grob.bound_details__right__text = right_markup
>>> text_spanner.override.text_spanner.dash_fraction = 1
>>> text_spanner.attach([staff])
>>> show(staff)
```



#### Bases

- `spannertools.Spanner`

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Spanner) .**components**

Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) .**leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) .**override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) .**set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Methods

(Spanner) .**append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
 Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
 Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
 Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
 Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
 Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) . **fracture** (*i*, *direction=None*)

Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) . **fuse** (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) . **get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) .**get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) .**index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) .**pop** ()

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) .**pop\_left** ()

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .**\_\_call\_\_** (*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) . **\_\_contains\_\_** (*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) . **\_\_copy\_\_** (\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) . **\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) . **\_\_getitem\_\_** (*expr*)

Gets item from spanner.

Returns component.

(Spanner) . **\_\_len\_\_** ()

Length of spanner.

Returns nonnegative integer.

(Spanner) . **\_\_lt\_\_** (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) . **\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

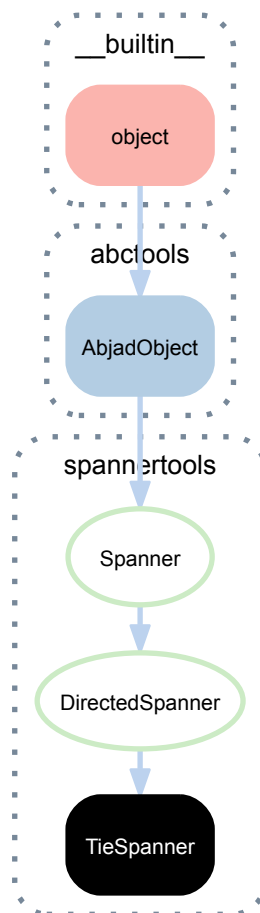
(Spanner) . **\_\_repr\_\_** ()

Interpreter representation of spanner.

Returns string.



### 32.2.22 spannertools.TieSpanner



**class** spannertools.**TieSpanner** (*music=None, direction=None, overrides=None*)

A tie spanner.

```
>>> staff = Staff(notetools.make_repeated_notes(4))
>>> spannertools.TieSpanner(staff[:])
TieSpanner(c'8, c'8, c'8, c'8)
```

```
>>> show(staff)
```



Returns tie spanner.

#### Bases

- spannertools.DirectedSpanner
- spannertools.Spanner
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

(Spanner).**components**  
Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**  
Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**  
LilyPond grob override component plug-in.  
Returns LilyPond grob override component plug-in.

(Spanner) **.set**  
LilyPond context setting component plug-in.  
Returns LilyPond context setting component plug-in.

## Read/write properties

(DirectedSpanner) **.direction**

## Methods

(Spanner) **.append** (*component*)  
Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.

(Spanner) **.append\_left** (*component*)  
Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
Attaches spanner to *components*.

Spanner must be empty.

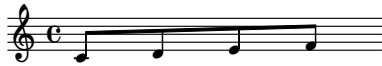
Returns none.

(Spanner) **.detach** ()  
Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) .**fracture** (*i*, *direction=None*)

Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) .**fuse** (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) .**get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) .**get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) .**index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) .**pop** ()

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) .**pop\_left** ()

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) .**\_\_call\_\_** (*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) .**\_\_contains\_\_**(*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) .**\_\_copy\_\_**(\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) .**\_\_getitem\_\_**(*expr*)

Gets item from spanner.

Returns component.

(Spanner) .**\_\_len\_\_**()

Length of spanner.

Returns nonnegative integer.

(Spanner) .**\_\_lt\_\_**(*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

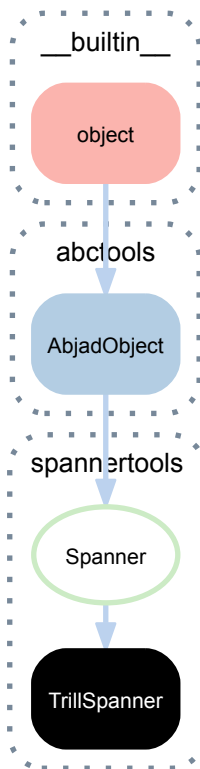
Returns boolean.

(Spanner) .**\_\_repr\_\_**()

Interpreter representation of spanner.

Returns string.

### 32.2.23 spannertools.TrillSpanner



**class** `spannertools.TrillSpanner` (*components=None, overrides=None*)  
 A trill spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.TrillSpanner(staff[:])
TrillSpanner(c'8, d'8, e'8, f'8)
```

```
>>> show(staff)
```



Override LilyPond TrillSpanner grob.

Returns trill spanner.

#### Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(Spanner).components`  
 Components in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Returns tuple.

(Spanner) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Spanner) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

## Read/write properties

TrillSpanner **.pitch**

Optional read / write pitch for pitched trills.

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> trill = spannertools.TrillSpanner(t[:2])
>>> trill.pitch = pitchtools.NamedPitch('cs', 4)
```

Set pitch.

TrillSpanner **.written\_pitch**

## Methods

(Spanner) **.append** (*component*)

Appends *component* to spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.append(voice[2])
>>> show(voice)
```



Returns none.



(Spanner) **.append\_left** (*component*)  
 Appends *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.append_left(voice[1])
>>> show(voice)
```



Returns none.

(Spanner) **.attach** (*components*)  
 Attaches spanner to *components*.

Spanner must be empty.

Returns none.

(Spanner) **.detach** ()  
 Detaches spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> spanner.detach()
>>> show(voice)
```



Returns none.

(Spanner) **.extend** (*components*)  
 Extends spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[:2])
>>> show(voice)
```



```
>>> spanner.extend(voice[2:])
>>> show(voice)
```



Returns none.

(Spanner) **.extend\_left** (*components*)  
 Extends left of spanner with *components*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.extend_left(voice[:2])
>>> show(voice)
```



Returns none.

(Spanner) . **fracture** (*i*, *direction=None*)

Fractures spanner at *direction* of component at index *i*.

Valid values for *direction* are Left, Right and None.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = spannertools.BeamSpanner(voice[:])
>>> show(voice)
```



```
>>> result = beam.fracture(1, direction=Left)
>>> show(voice)
```



```
>>> for x in result:
...     x
BeamSpanner(c'8, d'8, e'8, f'8)
BeamSpanner(c'8)
BeamSpanner(d'8, e'8, f'8)
```

Set *direction=None* to fracture on both left and right sides.

Returns tuple of original, left and right spanners.

(Spanner) . **fuse** (*spanner*)

Fuses spanner with contiguous *spanner*.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = spannertools.BeamSpanner(voice[:2])
>>> right_beam = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> result = left_beam.fuse(right_beam)
>>> show(voice)
```



```
>>> for x in result[0]:
...     x
BeamSpanner(c'8, d'8)
BeamSpanner(e'8, f'8)
BeamSpanner(c'8, d'8, e'8, f'8)
```

Returns list of left, right and new spanners.

(Spanner) . **get\_duration** (*in\_seconds=False*)

Gets duration of spanner.

Returns duration.

(Spanner) **.get\_timespan** (*in\_seconds=False*)

Gets timespan of spanner.

Returns timespan.

(Spanner) **.index** (*component*)

Returns index of *component* in spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.BeamSpanner(voice[2:])
>>> show(voice)
```



```
>>> spanner.index(voice[-2])
0
```

Returns nonnegative integer.

(Spanner) **.pop** ()

Pops rightmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop()
>>> show(voice)
```



Returns component.

(Spanner) **.pop\_left** ()

Pops leftmost component off of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> show(voice)
```



```
>>> note = spanner.pop_left()
>>> show(voice)
```



Returns component.

## Special methods

(Spanner) **.\_\_call\_\_** (*expr*)

Calls spanner on *expr*.

Same as attach.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> show(staff)
```



```
>>> beam = spannertools.BeamSpanner()
>>> beam = beam(staff[:])
>>> show(staff)
```



The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Returns spanner.

(Spanner) . **\_\_contains\_\_** (*expr*)

True when spanner contains *expr*. Otherwise false.

Returns boolean.

(Spanner) . **\_\_copy\_\_** (\**args*)

Copies spanner.

Does not copy spanner components.

Returns new spanner.

(AbjadObject) . **\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Spanner) . **\_\_getitem\_\_** (*expr*)

Gets item from spanner.

Returns component.

(Spanner) . **\_\_len\_\_** ()

Length of spanner.

Returns nonnegative integer.

(Spanner) . **\_\_lt\_\_** (*expr*)

True when spanner is less than *expr*.

Trivial comparison to allow doctests to work.

Returns boolean.

(AbjadObject) . **\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Spanner) . **\_\_repr\_\_** ()

Interpreter representation of spanner.

Returns string.

## 32.3 Functions

### 32.3.1 `spannertools.make_dynamic_spanner_below_with_nib_at_right`

`spannertools.make_dynamic_spanner_below_with_nib_at_right` (*dynamic\_text*, *components=None*)

Span *components* with text spanner. Position spanner below staff and configure with *dynamic\_text*, solid line and upward-pointing nib at right:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.make_dynamic_spanner_below_with_nib_at_right('mp', staff[:])
TextSpanner(c'8, d'8, e'8, f'8)
```

```
>>> show(staff)
```



Returns spanner.

### 32.3.2 `spannertools.make_solid_text_spanner_with_nib`

`spannertools.make_solid_text_spanner_with_nib` (*left\_text*, *components=None*, *direction=Up*)

Span *components* with solid line text spanner. Configure with *left\_text* and nib at right.

**Example 1.** Spanner above with downward-pointing nib:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.make_solid_text_spanner_with_nib(
...     'foo', staff[:], direction=Up)
>>> show(staff)
```



**Example 2.** Spanner below with upward-pointing nib:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.make_solid_text_spanner_with_nib(
...     'foo', staff[:], direction=Down)
>>> show(staff)
```



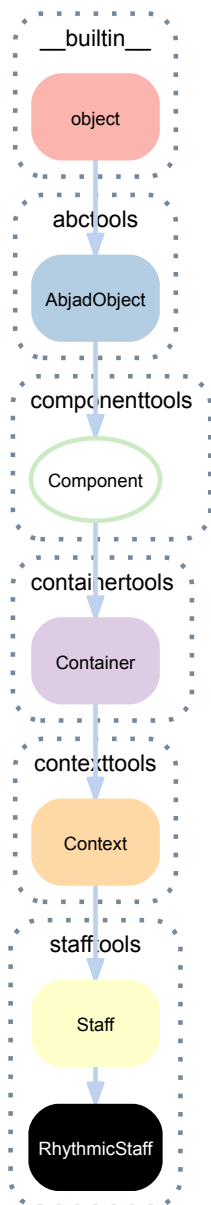
Returns spanner.



# STAFFTOOLS

## 33.1 Concrete classes

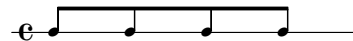
### 33.1.1 stafftools.RhythmicStaff



**class** `stafftools.RhythmicStaff` (*music=None, context\_name='RhythmicStaff', name=None*)  
 A rhythmic staff.

```
>>> staff = stafftools.RhythmicStaff("c'8 d'8 e'8 f'8")
```

```
>>> show(staff)
```



Returns `RhythmicStaff` instance.

## Bases

- `stafftools.Staff`
- `contexttools.Context`
- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Context) **.engraver\_consists**

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

(Context) **.engraver\_removals**

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

(Context) **.is\_semantic**

(Context) **.lilypond\_format**

(Component) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Component) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.



(Component).**storage\_format**

Storage format of component.

Returns string.

## Read/write properties

(Context).**context\_name**

Read / write name of context as a string.

(Context).**is\_nonsemantic**

Set indicator of nonsemantic voice:

```
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Returns boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interaction and other functions.

(Container).**is\_simultaneous**

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

(Context) **.name**

Read-write name of context. Must be string or none.

## Methods

(Container) **.append** (*component*)

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) **.extend** (*expr*)

Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```

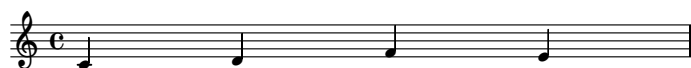


Returns none.

(Container) **.index** (*component*)

Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

(Container) **.pop** (*i=-1*)  
 Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

(Container) .**remove** (component)

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) .**reverse** ()

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) .**select** (sequential=False)

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

(Container) .**select\_leaves** (start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False)

Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) .**select\_notes\_and\_chords** ()

Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container) **.\_\_contains\_\_** (*expr*)

True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component) **.\_\_copy\_\_** (\*args)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(Container) **.\_\_delitem\_\_** (*i*)

Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Returns none.

(AbjadObject) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container) **.\_\_getitem\_\_** (*i*)

Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.

(Container) **.\_\_len\_\_** ()

Number of items in container.

Returns nonnegative integer.

(Component) **.\_\_mul\_\_** (*n*)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Context) **.\_\_repr\_\_** ()

(Component) **.\_\_rmul\_\_** (*n*)

Copies component *n* times and detach spanners.

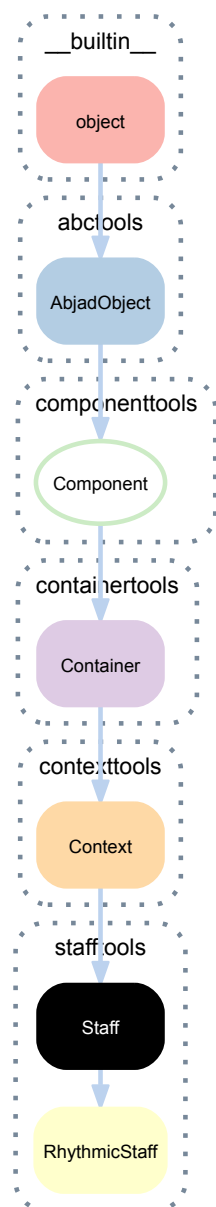
Returns list of new components.

(Container) **.\_\_setitem\_\_** (*i*, *expr*)

Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

Returns none.

### 33.1.2 stafftools.Staff



**class** `stafftools.Staff` (*music=None, context\_name='Staff', name=None*)  
 A staff.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> show(staff)
```



Returns Staff instance.

#### Bases

- `contexttools.Context`
- `containertools.Container`
- `componenttools.Component`

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `(Context).engraver_consists`

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

### `(Context).engraver_removals`

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

### `(Context).is_semantic`

### `(Context).lilypond_format`

### `(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

### `(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

### `(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

### `(Context).context_name`

Read / write name of context as a string.

### `(Context).is_nonsemantic`

Set indicator of nonsemantic voice:

```
>>> measures = \
...   measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Returns boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interaction and other functions.

(Container) **.is\_simultaneous**

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

(Context) **.name**

Read-write name of context. Must be string or none.

## Methods

(Container) **.append** (*component*)

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```





```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) **.extend** (*expr*)  
Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



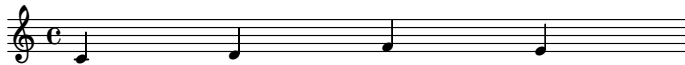
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) **.index** (*component*)  
Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

`(Container) .pop(i=-1)`

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

`(Container) .remove(component)`

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse()**  
Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (*sequential=False*)  
Selects component.  
Returns component selection when *sequential* is false.  
Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)  
Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords()**  
Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container) **.\_\_contains\_\_** (*expr*)  
True when *expr* appears in container. Otherwise false.  
Returns boolean.

(Component) **.\_\_copy\_\_** (*\*args*)  
Copies component with marks but without children of component or spanners attached to component.  
Returns new component.

(Container) **.\_\_delitem\_\_** (*i*)  
Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).  
Returns none.

(AbjadObject) .**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(Container) .**\_\_getitem\_\_**(*i*)  
 Get container *i*. Shallow traversal of container for numeric indices only.  
 Returns component.

(Container) .**\_\_len\_\_**()  
 Number of items in container.  
 Returns nonnegative integer.

(Component) .**\_\_mul\_\_**(*n*)  
 Copies component *n* times and detaches spanners.  
 Returns list of new components.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(Context) .**\_\_repr\_\_**()

(Component) .**\_\_rmul\_\_**(*n*)  
 Copies component *n* times and detach spanners.  
 Returns list of new components.

(Container) .**\_\_setitem\_\_**(*i, expr*)  
 Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.  
 Returns none.

## 33.2 Functions

### 33.2.1 stafftools.make\_rhythmic\_sketch\_staff

`stafftools.make_rhythmic_sketch_staff`(*music*)  
 Make rhythmic staff with transparent time\_signature and transparent bar lines.

# STRINGTOOLS

## 34.1 Functions

### 34.1.1 `stringtools.add_terminal_newlines`

`stringtools.add_terminal_newlines` (*lines*)

Add terminal newlines to *lines*:

```
>>> lines = ['first line', 'second line']
>>> stringtools.add_terminal_newlines(lines)
['first line\n', 'second line\n']
```

Do nothing when line in *lines* already ends in newline:

```
>>> lines = ['first line\n', 'second line\n']
>>> stringtools.add_terminal_newlines(lines)
['first line\n', 'second line\n']
```

Returns newly constructed object of *lines* type.

### 34.1.2 `stringtools.arg_to_bidirectional_direction_string`

`stringtools.arg_to_bidirectional_direction_string` (*arg*)

Convert *arg* to bidirectional direction string:

```
>>> stringtools.arg_to_bidirectional_direction_string('^')
'up'
```

```
>>> stringtools.arg_to_bidirectional_direction_string('_')
'down'
```

```
>>> stringtools.arg_to_bidirectional_direction_string(1)
'up'
```

```
>>> stringtools.arg_to_bidirectional_direction_string(-1)
'down'
```

If *arg* is 'up' or 'down', *arg* will be returned.

Returns string or none.

### 34.1.3 `stringtools.arg_to_bidirectional_lilypond_symbol`

`stringtools.arg_to_bidirectional_lilypond_symbol` (*arg*)

Convert *arg* to bidirectional LilyPond symbol:

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(Up)
'^'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(Down)
'_'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(1)
'^'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(-1)
'_'
```

If *arg* is '^' or '\_', *arg* will be returned.

Returns str or None.

### 34.1.4 stringtools.arg\_to\_tridirectional\_direction\_string

`stringtools.arg_to_tridirectional_direction_string(arg)`

Convert *arg* to tridirectional direction string:

```
>>> stringtools.arg_to_tridirectional_direction_string('^')
'up'
```

```
>>> stringtools.arg_to_tridirectional_direction_string('-')
'center'
```

```
>>> stringtools.arg_to_tridirectional_direction_string('_')
'down'
```

```
>>> stringtools.arg_to_tridirectional_direction_string(1)
'up'
```

```
>>> stringtools.arg_to_tridirectional_direction_string(0)
'center'
```

```
>>> stringtools.arg_to_tridirectional_direction_string(-1)
'down'
```

```
>>> stringtools.arg_to_tridirectional_direction_string('default')
'center'
```

If *arg* is None, None will be returned.

Returns str or None.

### 34.1.5 stringtools.arg\_to\_tridirectional\_lilypond\_symbol

`stringtools.arg_to_tridirectional_lilypond_symbol(arg)`

Convert *arg* to tridirectional LilyPond symbol:

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(Up)
'^'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol('neutral')
'_'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol('default')
'_'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(Down)
'_'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(1)
'^'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(0)
'_'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(-1)
'_'
```

If *arg* is None, None will be returned.

If *arg* is '^', '-', or '\_', *arg* will be returned.

Returns string or None.

### 34.1.6 stringtools.arg\_to\_tridirectional\_ordinal\_constant

`stringtools.arg_to_tridirectional_ordinal_constant(arg)`

Convert *arg* to tridirectional ordinal constant:

```
>>> stringtools.arg_to_tridirectional_ordinal_constant('^')
Up
```

```
>>> stringtools.arg_to_tridirectional_ordinal_constant('_')
Down
```

```
>>> stringtools.arg_to_tridirectional_ordinal_constant(1)
Up
```

```
>>> stringtools.arg_to_tridirectional_ordinal_constant(-1)
Down
```

If *arg* is Up, Center or Down, *arg* will be returned.

Returns OrdinalConstant or None.

### 34.1.7 stringtools.capitalize\_string\_start

`stringtools.capitalize_string_start(string)`

Capitalize *string*:

```
>>> string = 'violin I'
```

```
>>> stringtools.capitalize_string_start(string)
'Violin I'
```

Function differs from built-in `string.capitalize()`.

This function affects only `string[0]` and leaves noninitial characters as-is.

Built-in `string.capitalize()` forces noninitial characters to lowercase.

```
>>> string.capitalize()
'Violin i'
```

Returns newly constructed string.

### 34.1.8 stringtools.format\_input\_lines\_as\_doc\_string

`stringtools.format_input_lines_as_doc_string(input_lines)`

Format *input\_lines* as doc string.

Format expressions intelligently.

Treat blank lines intelligently.

Capture hash-suffixed line output.

Use when writing docstrings.

Example skipped because docstring goes crazy on example input.

### 34.1.9 `stringtools.format_input_lines_as_regression_test`

`stringtools.format_input_lines_as_regression_test` (*input\_lines*, *tab\_width*=3)

Format *input\_lines* as regression test:

```
>>> input_lines = '''
... staff = Staff("c'8 d'8 e'8 f'8")
... spannertools.BeamSpanner(staff.select_leaves())
... f(staff)
...
... tuplettools.FixedDurationTuplet(Duration(2, 8), staff[:3])
... f(staff)
... '''
```

```
>>> stringtools.format_input_lines_as_regression_test(input_lines)

staff = Staff("c'8 d'8 e'8 f'8")
spannertools.BeamSpanner(staff.select_leaves())

r'''
\\new Staff {
  c'8 [
    d'8
    e'8
    f'8 ]
}
'''

tuplettools.FixedDurationTuplet(Duration(2, 8), staff[:3])

r'''
\\new Staff {
  \\times 2/3 {
    c'8 [
      d'8
      e'8
    ]
  }
  f'8 ]
}

assert select(staff).is_well_formed()
assert staff.lilypond_format == "\\new Staff {
  \\n\\t\\times 2/3 {\\n\\t\\tc'8 [\\n\\t\\td'8\\n\\t\\te'8\\n\\t}\\n\\tf'8 ]\\n}"
'''
```

Format expressions intelligently.

Treat blank lines intelligently.

Remove line-final hash characters.

Used when writing tests.

### 34.1.10 `stringtools.is_dash_case_file_name`

`stringtools.is_dash_case_file_name` (*expr*)

True when *expr* is a string and is hyphen-delimited lowercase file name with extension:

```
>>> stringtools.is_dash_case_file_name('foo-bar')
True
```



False otherwise:

```
>>> stringtools.is_dash_case_file_name('foo.bar.blah')
False
```

Returns boolean.

### 34.1.11 stringtools.is\_dash\_case\_string

`stringtools.is_dash_case_string(expr)`

True when *expr* is a string and is hyphen delimited lowercase:

```
>>> stringtools.is_dash_case_string('foo-bar')
True
```

False otherwise:

```
>>> stringtools.is_dash_case_string('foo bar')
False
```

Returns boolean.

### 34.1.12 stringtools.is\_lower\_camel\_case\_string

`stringtools.is_lower_camel_case_string(expr)`

True when *expr* is a string and is lowercamelcase:

```
>>> stringtools.is_lower_camel_case_string('fooBar')
True
```

False otherwise:

```
>>> stringtools.is_lower_camel_case_string('FooBar')
False
```

Returns boolean.

### 34.1.13 stringtools.is\_snake\_case\_file\_name

`stringtools.is_snake_case_file_name(expr)`

True when *expr* is a string and is underscore-delimited lowercase file name with extension:

```
>>> stringtools.is_snake_case_file_name('foo_bar')
True
```

False otherwise:

```
>>> stringtools.is_snake_case_file_name('foo.bar.blah')
False
```

Returns boolean.

### 34.1.14 stringtools.is\_snake\_case\_file\_name\_with\_extension

`stringtools.is_snake_case_file_name_with_extension(expr)`

True when *expr* is a string and is underscore-delimited lowercase file name with extension:

```
>>> stringtools.is_snake_case_file_name_with_extension('foo_bar.blah')
True
```

False otherwise:

```
>>> stringtools.is_snake_case_file_name_with_extension('foo.bar.blah')
False
```

Returns boolean.

### 34.1.15 stringtools.is\_snake\_case\_package\_name

`stringtools.is_snake_case_package_name(expr)`

True when *expr* is a string and is underscore-delimited lowercase package name:

```
>>> stringtools.is_snake_case_package_name('foo.bar.blah_package')
True
```

False otherwise:

```
>>> stringtools.is_snake_case_package_name('foo.bar.BlahPackage')
False
```

Returns boolean.

### 34.1.16 stringtools.is\_snake\_case\_string

`stringtools.is_snake_case_string(expr)`

True when *expr* is a string and is underscore delimited lowercase:

```
>>> stringtools.is_snake_case_string('foo_bar')
True
```

False otherwise:

```
>>> stringtools.is_snake_case_string('foo bar')
False
```

Returns boolean.

### 34.1.17 stringtools.is\_space\_delimited\_lowercase\_string

`stringtools.is_space_delimited_lowercase_string(expr)`

True when *expr* is a string and is space-delimited lowercase:

```
>>> stringtools.is_space_delimited_lowercase_string('foo bar')
True
```

False otherwise:

```
>>> stringtools.is_space_delimited_lowercase_string('foo_bar')
False
```

Returns boolean.

### 34.1.18 stringtools.is\_upper\_camel\_case\_string

`stringtools.is_upper_camel_case_string(expr)`

True when *expr* is a string and is uppercamelcase:

```
>>> stringtools.is_upper_camel_case_string('FooBar')
True
```

False otherwise:

```
>>> stringtools.is_upper_camel_case_string('fooBar')
False
```

Returns boolean.

### 34.1.19 stringtools.pluralize\_string

`stringtools.pluralize_string(string)`  
Pluralize English *string*. Change terminal -y to -ies:

```
>>> stringtools.pluralize_string('catenary')
'catenaries'
```

Add -es to terminal -s, -sh, -x and -z:

```
>>> stringtools.pluralize_string('brush')
'brushes'
```

Add -s to all other strings:

```
>>> stringtools.pluralize_string('shape')
'shapes'
```

Returns string.

### 34.1.20 stringtools.snake\_case\_to\_lower\_camel\_case

`stringtools.snake_case_to_lower_camel_case(string)`  
Change underscore-delimited lowercase *string* to lowercamelcase:

```
>>> string = 'bass_figure_alignment_positioning'
>>> stringtools.snake_case_to_lower_camel_case(string)
'bassFigureAlignmentPositioning'
```

Returns string.

### 34.1.21 stringtools.snake\_case\_to\_upper\_camel\_case

`stringtools.snake_case_to_upper_camel_case(string)`  
Change underscore-delimited lowercase *string* to uppercamelcase:

```
>>> string = 'bass_figure_alignment_positioning'
>>> stringtools.snake_case_to_upper_camel_case(string)
'BassFigureAlignmentPositioning'
```

Returns string.

### 34.1.22 stringtools.space\_delimited\_lowercase\_to\_upper\_camel\_case

`stringtools.space_delimited_lowercase_to_upper_camel_case(string)`  
Change space-delimited lowercase *string* to uppercamelcase:

```
>>> string = 'bass figure alignment positioning'
>>> stringtools.space_delimited_lowercase_to_upper_camel_case(string)
'BassFigureAlignmentPositioning'
```

Returns string.

### 34.1.23 stringtools.string\_to\_accent\_free\_snake\_case

`stringtools.string_to_accent_free_snake_case(string)`  
Change *string* to strict directory name:

```
>>> stringtools.string_to_accent_free_snake_case('Déja vu')
'deja_vu'
```

Strip accents from accented characters. Change all punctuation (including spaces) to underscore. Set to lowercase.

Returns string.

### 34.1.24 stringtools.string\_to\_space\_delimited\_lowercase

`stringtools.string_to_space_delimited_lowercase(string)`

Change uppercamelcase *string* to space-delimited lowercase:

```
>>> stringtools.string_to_space_delimited_lowercase('TieSpanner')
'tie spanner'
```

Change underscore-delimited *string* to space-delimited lowercase:

```
>>> stringtools.string_to_space_delimited_lowercase('tie_spanner')
'tie spanner'
```

Returns space-delimited string unchanged:

```
>>> stringtools.string_to_space_delimited_lowercase('tie spanner')
'tie spanner'
```

Returns empty *string* unchanged:

```
>>> stringtools.string_to_space_delimited_lowercase('')
''
```

Returns string.

### 34.1.25 stringtools.strip\_diacritics\_from\_binary\_string

`stringtools.strip_diacritics_from_binary_string(binary_string)`

Strip diacritics from *binary\_string*:

```
>>> binary_string = 'Dvo\x5c\x99\x3\xalk'
```

```
>>> print binary_string
Dvořák
```

```
>>> stringtools.strip_diacritics_from_binary_string(binary_string)
'Dvorak'
```

Returns ASCII string.

### 34.1.26 stringtools.upper\_camel\_case\_to\_snake\_case

`stringtools.upper_camel_case_to_snake_case(string)`

Change uppercamelcase *string* to underscore-delimited lowercase:

```
>>> string = 'KeySignatureMark'
```

```
>>> stringtools.upper_camel_case_to_snake_case(string)
'key_signature_mark'
```

Returns string.

### 34.1.27 stringtools.upper\_camel\_case\_to\_space\_delimited\_lowercase

stringtools.**upper\_camel\_case\_to\_space\_delimited\_lowercase**(*string*)

Change uppercamelcase *string* to space-delimited lowercase:

```
>>> string = 'KeySignatureMark'
```

```
>>> stringtools.upper_camel_case_to_space_delimited_lowercase(string)
'key signature mark'
```

Returns string.



# TEMPOTOOLS

## 35.1 Functions

### 35.1.1 tempotools.report\_integer\_tempo\_rewrite\_pairs

`tempotools.report_integer_tempo_rewrite_pairs` (*integer\_tempo*, *maximum\_numerator*=None, *maximum\_denominator*=None) *maxi-*

Report *integer\_tempo* rewrite pairs.

Allow no tempo less than half *integer\_tempo* or greater than double *integer\_tempo*:

```
>>> tempotools.report_integer_tempo_rewrite_pairs(  
...     58, maximum_numerator=8, maximum_denominator=8)  
2:1      29  
1:1      58  
2:3      87  
1:2     116
```

With more lenient numerator and denominator:

```
>>> tempotools.report_integer_tempo_rewrite_pairs(  
...     58, maximum_numerator=30, maximum_denominator=30)  
2:1      29  
29:15     30  
29:16     32  
29:17     34  
29:18     36  
29:19     38  
29:20     40  
29:21     42  
29:22     44  
29:23     46  
29:24     48  
29:25     50  
29:26     52  
29:27     54  
29:28     56  
1:1      58  
29:30     60  
2:3      87  
1:2     116
```

Returns none.

### 35.1.2 tempotools.rewrite\_duration\_under\_new\_tempo

`tempotools.rewrite_duration_under_new_tempo` (*duration*, *tempo\_mark\_1*, *tempo\_mark\_2*)

Rewrite prolated *duration* under new tempo.

Given prolated *duration* governed by *tempo\_mark\_1*, return new duration governed by *tempo\_mark\_2*.

Ensure that *duration* and new duration consume exactly the same amount of time in seconds.

**Example.** Consider the two tempo indications below.

```
>>> tempo_mark_1 = contexttools.TempoMark(Duration(1, 4), 60)
>>> tempo_mark_2 = contexttools.TempoMark(Duration(1, 4), 90)
```

The first tempo indication specifies quarter equal to 60 MM.

The second tempo indication specifies quarter equal to 90 MM.

The second tempo is  $3/2$  times as fast as the first:

```
>>> tempo_mark_2 / tempo_mark_1
Multiplier(3, 2)
```

Note that a triplet eighth note *tempo\_mark\_1* equals a regular eighth note under *tempo\_mark\_2*:

```
>>> tempotools.rewrite_duration_under_new_tempo(
...     Duration(1, 12), tempo_mark_1, tempo_mark_2)
Duration(1, 8)
```

And note that a regular eighth note under *tempo\_mark\_1* equals a dotted sixteenth under *tempo\_mark\_2*:

```
>>> tempotools.rewrite_duration_under_new_tempo(
...     Duration(1, 8), tempo_mark_1, tempo_mark_2)
Duration(3, 16)
```

Returns duration.

### 35.1.3 tempotools.rewrite\_integer\_tempo

`tempotools.rewrite_integer_tempo(integer_tempo, maximum_numerator=None, maximum_denominator=None)`

Rewrite *integer\_tempo*.

Allow no tempo less than half *integer\_tempo* or greater than double *integer\_tempo*:

```
>>> pairs = tempotools.rewrite_integer_tempo(
...     58, maximum_numerator=8, maximum_denominator=8)
```

```
>>> for pair in pairs:
...     pair
...
(Multiplier(1, 2), 29)
(Multiplier(1, 1), 58)
(Multiplier(3, 2), 87)
(Multiplier(2, 1), 116)
```

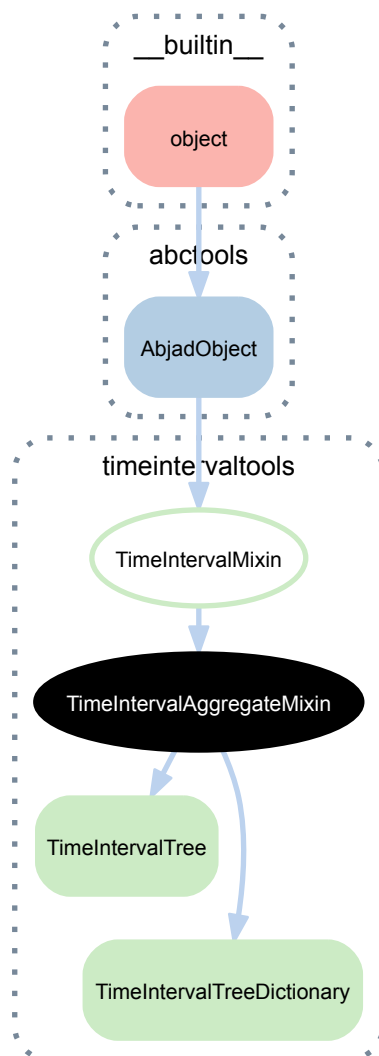
Returns list.



# TIMEINTERVALTOOLS

## 36.1 Abstract classes

### 36.1.1 `timeintervaltools.TimeIntervalAggregateMixin`



**class** `timeintervaltools.TimeIntervalAggregateMixin`  
A time-interval aggregate mixin.

## Bases

- `timeintervaltools.TimeIntervalMixin`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`TimeIntervalAggregateMixin.all_unique_bounds`

`(TimeIntervalMixin).bounds`

Start and stop\_offset of self returned as `TimeInterval` instance:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

`(TimeIntervalMixin).center`

Center offset of start\_offset and stop\_offset offsets:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.center
Offset(6, 1)
```

Returns *Offset* instance.

`(TimeIntervalMixin).duration`

Duration of the time interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.duration
Duration(8, 1)
```

Returns *Duration* instance.

`TimeIntervalAggregateMixin.earliest_start`

`TimeIntervalAggregateMixin.earliest_stop`

`TimeIntervalAggregateMixin.intervals`

`TimeIntervalAggregateMixin.latest_start`

`TimeIntervalAggregateMixin.latest_stop`

`TimeIntervalAggregateMixin.offset_counts`

`TimeIntervalAggregateMixin.offsets`

`(TimeIntervalMixin).signature`

Tuple of start\_offset bound and stop\_offset bound.

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.signature
(Offset(2, 1), Offset(10, 1))
```

Returns 2-tuple of *Offset* instances.

`(TimeIntervalMixin).start_offset`

Starting offset of interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.start_offset
Offset(2, 1)
```

Returns *Offset* instance.

`(TimeIntervalMixin).stop_offset`

Stopping offset of interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.stop_offset
Offset(10, 1)
```

Returns *Offset* instance.

`TimeIntervalAggregateMixin.storage_format`

Storage format of time interval aggregate mixin.

Returns string.

## Methods

`TimeIntervalAggregateMixin.calculate_attack_density(bounding_interval=None)`

`TimeIntervalAggregateMixin.calculate_depth_centroid(bounding_interval=None)`

Calculate the weighted mean offset of *intervals*, such that the centroids of each interval in the depth tree of *intervals* make up the values of the mean, and the depth of each interval in the depth tree of *intervals* make up the weights.

Returns *Offset*.

`TimeIntervalAggregateMixin.calculate_depth_density(bounding_interval=None)`

Returns a fraction, of the duration of each interval in the depth tree of *intervals*, multiplied by the depth at that interval, divided by the overall duration of *intervals*.

The depth density of a single interval is *I*:

Returns multiplier.

`TimeIntervalAggregateMixin.calculate_mean_attack_offset()`

`TimeIntervalAggregateMixin.calculate_mean_release_offset()`

`TimeIntervalAggregateMixin.calculate_minimum_mean_and_maximum_depths()`

Returns a 3-tuple of the minimum, mean and maximum depth of *intervals*.

If *intervals* is empty, return *None*.

“Mean” in this case is a weighted mean, where the durations of the intervals in depth tree of *intervals* are the weights.

`TimeIntervalAggregateMixin.calculate_minimum_mean_and_maximum_durations()`

Returns a 3-tuple of the minimum, mean and maximum duration of all intervals in *intervals*.

If *intervals* is empty, return *None*.

`TimeIntervalAggregateMixin.calculate_release_density(bounding_interval=None)`

`TimeIntervalAggregateMixin.calculate_sustain_centroid()`

Returns a weighted mean, such that the centroid of each interval in *intervals* are the values, and the weights are their durations.

`TimeIntervalAggregateMixin.clip_interval_durations_to_range(minimum=None, maximum=None)`

`TimeIntervalAggregateMixin.compute_depth(bounding_interval=None)`

Compute a tree whose intervals represent the level of overlap of the time interval aggregate:

```
>>> a = timeintervaltools.TimeInterval(0, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
>>> c = timeintervaltools.TimeInterval(9, 15)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> tree.compute_depth()
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'depth': 1}),
    TimeInterval(Offset(3, 1), Offset(6, 1), {'depth': 0}),
    TimeInterval(Offset(6, 1), Offset(9, 1), {'depth': 1}),
    TimeInterval(Offset(9, 1), Offset(12, 1), {'depth': 2}),
    TimeInterval(Offset(12, 1), Offset(15, 1), {'depth': 1})
])
```

If *bounding\_interval* is not none, only consider the depth of time intervals which intersect that time interval:

```
>>> a = timeintervaltools.TimeInterval(0, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
>>> c = timeintervaltools.TimeInterval(9, 15)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> d = timeintervaltools.TimeInterval(-1, 16)
>>> tree.compute_depth(bounding_interval=d)
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(0, 1), {'depth': 0}),
    TimeInterval(Offset(0, 1), Offset(3, 1), {'depth': 1}),
    TimeInterval(Offset(3, 1), Offset(6, 1), {'depth': 0}),
    TimeInterval(Offset(6, 1), Offset(9, 1), {'depth': 1}),
    TimeInterval(Offset(9, 1), Offset(12, 1), {'depth': 2}),
    TimeInterval(Offset(12, 1), Offset(15, 1), {'depth': 1}),
    TimeInterval(Offset(15, 1), Offset(16, 1), {'depth': 0})
])
```

Returns interval tree.

`TimeIntervalAggregateMixin.compute_logical_and` (*bounding\_interval=None*)  
Compute logical AND of intervals.

Returns time interval tree.

`TimeIntervalAggregateMixin.compute_logical_not` (*bounding\_interval=None*)  
Compute logical NOT of intervals.

Returns time interval tree.

`TimeIntervalAggregateMixin.compute_logical_or` (*bounding\_interval=None*)  
Compute logical OR of intervals.

Returns time interval tree.

`TimeIntervalAggregateMixin.compute_logical_xor` (*bounding\_interval=None*)  
Compute logical XOR of intervals.

Returns time interval tree.

`TimeIntervalAggregateMixin.find_intervals_intersecting_or_tangent_to_interval` ()

`TimeIntervalAggregateMixin.find_intervals_intersecting_or_tangent_to_offset` ()

`TimeIntervalAggregateMixin.find_intervals_starting_after_offset` ()

`TimeIntervalAggregateMixin.find_intervals_starting_and_stopping_within_interval` ()

`TimeIntervalAggregateMixin.find_intervals_starting_at_offset` ()

`TimeIntervalAggregateMixin.find_intervals_starting_before_offset` ()

`TimeIntervalAggregateMixin.find_intervals_starting_or_stopping_at_offset` ()

`TimeIntervalAggregateMixin.find_intervals_starting_within_interval` ()

`TimeIntervalAggregateMixin.find_intervals_stopping_after_offset` ()

`TimeIntervalAggregateMixin.find_intervals_stopping_at_offset` ()

```

TimeIntervalAggregateMixin.find_intervals_stopping_before_offset()
TimeIntervalAggregateMixin.find_intervals_stopping_within_interval()
TimeIntervalAggregateMixin.fuse_overlapping_intervals (include_tangent_intervals=False)
(TimeIntervalMixin).get_overlap_with_interval (interval)
    Returns amount of overlap with interval.
(TimeIntervalMixin).is_contained_by_interval (interval)
    True if interval is contained by interval.
(TimeIntervalMixin).is_container_of_interval (interval)
    True if interval contains interval.
(TimeIntervalMixin).is_overlapped_by_interval (interval)
    True if interval is overlapped by interval.
(TimeIntervalMixin).is_tangent_to_interval (interval)
    True if interval is tangent to interval.
TimeIntervalAggregateMixin.partition (include_tangent_intervals=False)
    Partition aggregate into groups of overlapping intervals:

```

```

>>> tree = timeintervaltools.TimeIntervalTree(
...     timeintervaltools.make_test_intervals())
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'}),
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'}),
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'}),
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'}),
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'}),
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])

```

```

>>> for group in tree.partition():
...     group
...
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'})
])
TimeIntervalTree([
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'})
])
TimeIntervalTree([
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'})
])
TimeIntervalTree([
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'})
])
TimeIntervalTree([
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'})
])
TimeIntervalTree([
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])

```

If *include\_tangent\_intervals* is true, treat tangent intervals as part of the same group:

```
>>> for group in tree.partition(include_tangent_intervals=True):
...     group
...
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'})
])
TimeIntervalTree([
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'})
])
TimeIntervalTree([
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'})
])
TimeIntervalTree([
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'})
])
TimeIntervalTree([
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'}),
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])
```

Returns 0 or more trees.

(TimeIntervalMixin).**quantize\_to\_rational**(*rational*)

(TimeIntervalMixin).**scale\_by\_rational**(*rational*)

TimeIntervalAggregateMixin.**scale\_interval\_durations\_by\_rational**(*rational*)

TimeIntervalAggregateMixin.**scale\_interval\_durations\_to\_rational**(*rational*)

TimeIntervalAggregateMixin.**scale\_interval\_offsets\_by\_rational**(*rational*)

(TimeIntervalMixin).**scale\_to\_rational**(*rational*)

(TimeIntervalMixin).**shift\_by\_rational**(*rational*)

(TimeIntervalMixin).**shift\_to\_rational**(*rational*)

(TimeIntervalMixin).**split\_at\_rationals**(\**rationals*)

## Special methods

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

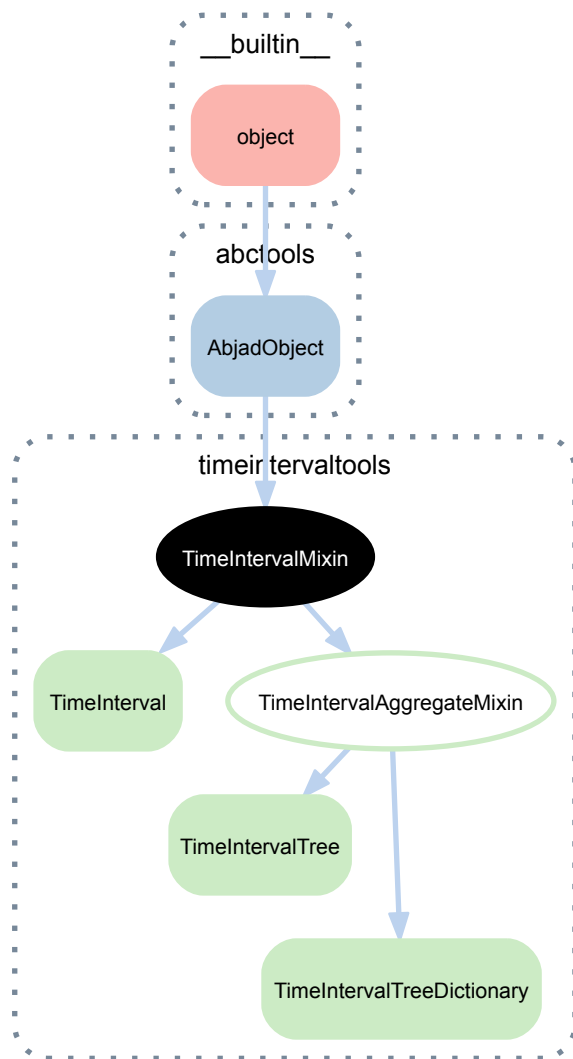
(TimeIntervalMixin).**\_\_nonzero\_\_**()

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

### 36.1.2 timeintervaltools.TimeIntervalMixin



**class** `timeintervaltools.TimeIntervalMixin`  
 A time-interval mixin.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`TimeIntervalMixin.bounds`

Start and stop\_offset of self returned as `TimeInterval` instance:

```

>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
  
```

Returns *TimeInterval* instance.

`TimeIntervalMixin.center`

Center offset of `start_offset` and `stop_offset` offsets:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.center
Offset(6, 1)
```

Returns *Offset* instance.

`TimeIntervalMixin.duration`

Duration of the time interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.duration
Duration(8, 1)
```

Returns *Duration* instance.

`TimeIntervalMixin.signature`

Tuple of `start_offset` bound and `stop_offset` bound.

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.signature
(Offset(2, 1), Offset(10, 1))
```

Returns 2-tuple of *Offset* instances.

`TimeIntervalMixin.start_offset`

Starting offset of interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.start_offset
Offset(2, 1)
```

Returns *Offset* instance.

`TimeIntervalMixin.stop_offset`

Stopping offset of interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.stop_offset
Offset(10, 1)
```

Returns *Offset* instance.

## Methods

`TimeIntervalMixin.get_overlap_with_interval(interval)`

Returns amount of overlap with *interval*.

`TimeIntervalMixin.is_contained_by_interval(interval)`

True if interval is contained by *interval*.

`TimeIntervalMixin.is_container_of_interval(interval)`

True if interval contains *interval*.

`TimeIntervalMixin.is_overlapped_by_interval(interval)`

True if interval is overlapped by *interval*.

`TimeIntervalMixin.is_tangent_to_interval(interval)`

True if interval is tangent to *interval*.

`TimeIntervalMixin.quantize_to_rational(rational)`

`TimeIntervalMixin.scale_by_rational(rational)`

`TimeIntervalMixin.scale_to_rational(rational)`

`TimeIntervalMixin.shift_by_rational(rational)`



`TimeIntervalMixin.shift_to_rational` (*rational*)

`TimeIntervalMixin.split_at_rationals` (*\*rationals*)

## Special methods

`(AbjadObject).__eq__` (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__` (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`TimeIntervalMixin.__nonzero__` ()

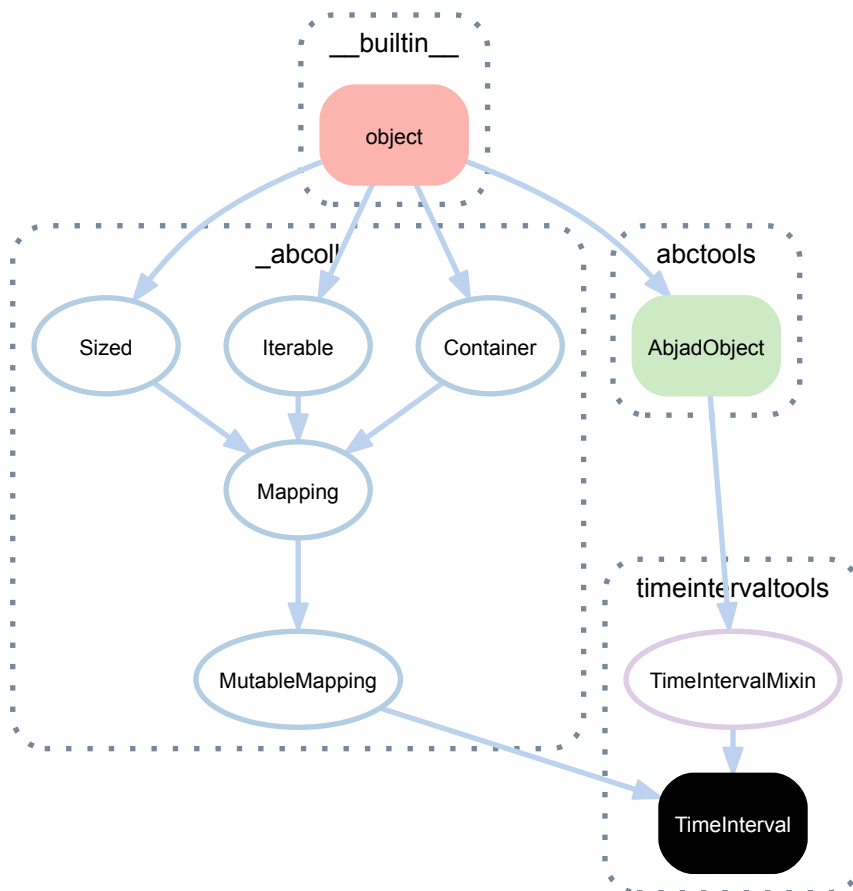
`(AbjadObject).__repr__` ()

Interpreter representation of Abjad object.

Returns string.

## 36.2 Concrete classes

### 36.2.1 timeintervaltools.TimeInterval



`class timeintervaltools.TimeInterval` (*\*args*)

A start\_offset / stop\_offset pair, carrying some metadata.

## Bases

- `timeintervaltools.TimeIntervalMixin`
- `abctools.AbjadObject`
- `_abcoll.MutableMapping`
- `_abcoll.Mapping`
- `_abcoll.Sized`
- `_abcoll.Iterable`
- `_abcoll.Container`
- `__builtin__.object`

## Read-only properties

(`TimeIntervalMixin`).**`bounds`**

Start and stop\_offset of self returned as `TimeInterval` instance:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

`TimeInterval`.**`center`**

Center point of start\_offset and stop\_offset bounds.

`TimeInterval`.**`duration`**

stop\_offset bound minus start\_offset bound.

`TimeInterval`.**`signature`**

Tuple of start\_offset bound and stop\_offset bound.

`TimeInterval`.**`start_offset`**

start\_offset bound.

`TimeInterval`.**`stop_offset`**

stop\_offset bound.

`TimeInterval`.**`storage_format`**

Storage format of time interval.

Returns string.

## Methods

(`MutableMapping`).**`clear()`** → None. Remove all items from D.

(`Mapping`).**`get(k[, d])`** → D[k] if k in D, else d. d defaults to None.

(`TimeIntervalMixin`).**`get_overlap_with_interval(interval)`**

Returns amount of overlap with *interval*.

(`TimeIntervalMixin`).**`is_contained_by_interval(interval)`**

True if interval is contained by *interval*.

(`TimeIntervalMixin`).**`is_container_of_interval(interval)`**

True if interval contains *interval*.

(TimeIntervalMixin) **.is\_overlapped\_by\_interval** (*interval*)  
 True if *interval* is overlapped by *interval*.

(TimeIntervalMixin) **.is\_tangent\_to\_interval** (*interval*)  
 True if *interval* is tangent to *interval*.

(Mapping) **.items** () → list of D's (key, value) pairs, as 2-tuples

(Mapping) **.iteritems** () → an iterator over the (key, value) items of D

(Mapping) **.iterkeys** () → an iterator over the keys of D

(Mapping) **.itervalues** () → an iterator over the values of D

(Mapping) **.keys** () → list of D's keys

(MutableMapping) **.pop** (*k*, *d*) → *v*, remove specified key and return the corresponding value.  
 If key is not found, *d* is returned if given, otherwise `KeyError` is raised.

(MutableMapping) **.popitem** () → (*k*, *v*), remove and return some (key, value) pair  
 as a 2-tuple; but raise `KeyError` if D is empty.

TimeInterval **.quantize\_to\_rational** (*rational*)

TimeInterval **.scale\_by\_rational** (*rational*)

TimeInterval **.scale\_to\_rational** (*rational*)

(MutableMapping) **.setdefault** (*k*, *d*) → *D.get(k,d)*, also set *D[k]=d* if *k* not in D

TimeInterval **.shift\_by\_rational** (*rational*)

TimeInterval **.shift\_to\_rational** (*rational*)

TimeInterval **.split\_at\_rationals** (*\*rationals*)

(MutableMapping) **.update** (*[E]*, *\*\*F*) → None. Update D from mapping/iterable E and F.  
 If E present and has a `.keys()` method, does: for *k* in E: *D[k] = E[k]* If E present and lacks `.keys()` method,  
 does: for (*k*, *v*) in E: *D[k] = v* In either case, this is followed by: for *k*, *v* in *F.items()*: *D[k] = v*

(Mapping) **.values** () → list of D's values

## Special methods

(Mapping) **.\_\_contains\_\_** (*key*)

TimeInterval **.\_\_delitem\_\_** (*item*)

TimeInterval **.\_\_eq\_\_** (*expr*)

TimeInterval **.\_\_getitem\_\_** (*item*)

TimeInterval **.\_\_hash\_\_** ()

TimeInterval **.\_\_iter\_\_** ()

TimeInterval **.\_\_len\_\_** ()

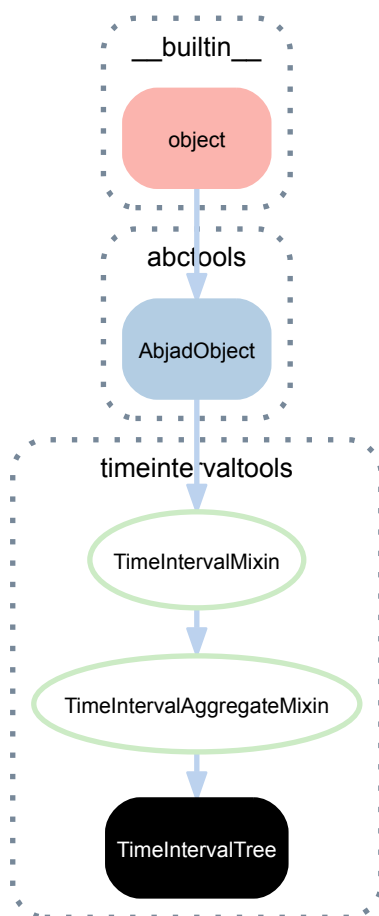
TimeInterval **.\_\_ne\_\_** (*expr*)

(TimeIntervalMixin) **.\_\_nonzero\_\_** ()

TimeInterval **.\_\_repr\_\_** ()

TimeInterval **.\_\_setitem\_\_** (*item*, *value*)

### 36.2.2 timeintervaltools.TimeIntervalTree



**class** timeintervaltools.**TimeIntervalTree** (*intervals=None*)

An augmented red-black tree for storing and searching for intervals of time (rather than pitch).

This allows for the arbitrary placement of blocks of material along a time-line. While this functionality could be achieved with Python’s built-in collections, this class reduces the complexity of the search process, such as locating overlapping intervals.

TimeIntervalTrees can be instantiated without contents, or from a mixed collection of other TimeIntervalTrees and / or TimeIntervals. The input will be parsed recursively:

```

>>> from abjad.tools.timeintervaltools import *

>>> interval_one = TimeInterval(0, 10)
>>> interval_two = TimeInterval(1, 8)
>>> interval_three = TimeInterval(3, 13)
>>> tree = TimeIntervalTree(
...     [interval_one, interval_two, interval_three])

>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(10, 1), {}),
    TimeInterval(Offset(1, 1), Offset(8, 1), {}),
    TimeInterval(Offset(3, 1), Offset(13, 1), {})
])
  
```

Return *TimeIntervalTree* instance.

#### Bases

- timeintervaltools.TimeIntervalAggregateMixin

- `timeintervaltools.TimeIntervalMixin`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TimeIntervalAggregateMixin).all_unique_bounds`

`(TimeIntervalMixin).bounds`

Start and stop\_offset of self returned as TimeInterval instance:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

`(TimeIntervalMixin).center`

Center offset of start\_offset and stop\_offset offsets:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.center
Offset(6, 1)
```

Returns *Offset* instance.

`TimeIntervalTree.duration`

Absolute difference of the stop\_offset and start\_offset values of the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.duration
Duration(5, 2)
```

Empty trees have a duration of 0.

Return *Duration* instance.

`TimeIntervalTree.earliest_start`

The minimum start\_offset value of all intervals in the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.earliest_start
Offset(1, 1)
```

Return *Offset* instance, or *None* if tree is empty.

`TimeIntervalTree.earliest_stop`

The minimum stop\_offset value of all intervals in the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.earliest_stop
Offset(2, 1)
```

Return *Offset* instance, or *None* if tree is empty.

`TimeIntervalTree.intervals`

`TimeIntervalTree.intervals_are_contiguous`

True when all intervals are contiguous and non-overlapping, otherwise False.

Returns boolean.

`TimeIntervalTree.intervals_are_nonoverlapping`

True when all intervals are non-overlapping, otherwise False.

Returns boolean.

`TimeIntervalTree.latest_start`

The maximum `start_offset` value of all intervals in the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.latest_start
Offset(3, 1)
```

Return `Offset` instance, or `None` if tree is empty.

`TimeIntervalTree.latest_stop`

The maximum `stop_offset` value of all intervals in the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.latest_stop
Offset(7, 2)
```

Return `Offset` instance, or `None` if tree is empty.

`(TimeIntervalAggregateMixin).offset_counts`

`(TimeIntervalAggregateMixin).offsets`

`(TimeIntervalMixin).signature`

Tuple of `start_offset` bound and `stop_offset` bound.

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.signature
(Offset(2, 1), Offset(10, 1))
```

Returns 2-tuple of *Offset* instances.

`(TimeIntervalMixin).start_offset`

Starting offset of interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.start_offset
Offset(2, 1)
```

Returns *Offset* instance.

`(TimeIntervalMixin).stop_offset`

Stopping offset of interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.stop_offset
Offset(10, 1)
```

Returns *Offset* instance.

`(TimeIntervalAggregateMixin).storage_format`

Storage format of time interval aggregate mixin.

Returns string.

## Methods

`(TimeIntervalAggregateMixin).calculate_attack_density(bounding_interval=None)`

`(TimeIntervalAggregateMixin).calculate_depth_centroid(bounding_interval=None)`

Calculate the weighted mean offset of *intervals*, such that the centroids of each interval in the depth tree of *intervals* make up the values of the mean, and the depth of each interval in the depth tree of *intervals* make up the weights.

Returns Offset.

`(TimeIntervalAggregateMixin).calculate_depth_density(bounding_interval=None)`

Returns a fraction, of the duration of each interval in the depth tree of *intervals*, multiplied by the depth at that interval, divided by the overall duration of *intervals*.

The depth density of a single interval is  $l$ :

Returns multiplier.

`(TimeIntervalAggregateMixin).calculate_mean_attack_offset()`

`(TimeIntervalAggregateMixin).calculate_mean_release_offset()`

`(TimeIntervalAggregateMixin).calculate_minimum_mean_and_maximum_depths()`

Returns a 3-tuple of the minimum, mean and maximum depth of *intervals*.

If *intervals* is empty, return None.

“Mean” in this case is a weighted mean, where the durations of the intervals in depth tree of *intervals* are the weights.

`(TimeIntervalAggregateMixin).calculate_minimum_mean_and_maximum_durations()`

Returns a 3-tuple of the minimum, mean and maximum duration of all intervals in *intervals*.

If *intervals* is empty, return None.

`(TimeIntervalAggregateMixin).calculate_release_density(bounding_interval=None)`

`(TimeIntervalAggregateMixin).calculate_sustain_centroid()`

Returns a weighted mean, such that the centroid of each interval in *intervals* are the values, and the weights are their durations.

`TimeIntervalTree.clip_interval_durations_to_range(minimum=None, maximum=None)`

`(TimeIntervalAggregateMixin).compute_depth(bounding_interval=None)`

Compute a tree whose intervals represent the level of overlap of the time interval aggregate:

```
>>> a = timeintervaltools.TimeInterval(0, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
>>> c = timeintervaltools.TimeInterval(9, 15)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> tree.compute_depth()
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'depth': 1}),
    TimeInterval(Offset(3, 1), Offset(6, 1), {'depth': 0}),
    TimeInterval(Offset(6, 1), Offset(9, 1), {'depth': 1}),
    TimeInterval(Offset(9, 1), Offset(12, 1), {'depth': 2}),
    TimeInterval(Offset(12, 1), Offset(15, 1), {'depth': 1})
])
```

If *bounding\_interval* is not none, only consider the depth of time intervals which intersect that time interval:

```
>>> a = timeintervaltools.TimeInterval(0, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
>>> c = timeintervaltools.TimeInterval(9, 15)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> d = timeintervaltools.TimeInterval(-1, 16)
>>> tree.compute_depth(bounding_interval=d)
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(0, 1), {'depth': 0}),
```

```
TimeInterval(Offset(0, 1), Offset(3, 1), {'depth': 1}),
TimeInterval(Offset(3, 1), Offset(6, 1), {'depth': 0}),
TimeInterval(Offset(6, 1), Offset(9, 1), {'depth': 1}),
TimeInterval(Offset(9, 1), Offset(12, 1), {'depth': 2}),
TimeInterval(Offset(12, 1), Offset(15, 1), {'depth': 1}),
TimeInterval(Offset(15, 1), Offset(16, 1), {'depth': 0})
])
```

Returns interval tree.

(TimeIntervalAggregateMixin).**compute\_logical\_and**(*bounding\_interval=None*)  
Compute logical AND of intervals.

Returns time interval tree.

(TimeIntervalAggregateMixin).**compute\_logical\_not**(*bounding\_interval=None*)  
Compute logical NOT of intervals.

Returns time interval tree.

(TimeIntervalAggregateMixin).**compute\_logical\_or**(*bounding\_interval=None*)  
Compute logical OR of intervals.

Returns time interval tree.

(TimeIntervalAggregateMixin).**compute\_logical\_xor**(*bounding\_interval=None*)  
Compute logical XOR of intervals.

Returns time interval tree.

TimeIntervalTree.**explode\_intervals**(*aggregate\_count=None*)  
Explode intervals into trees, avoiding overlap and distributing density.

```
>>> tree = timeintervaltools.TimeIntervalTree(
...     timeintervaltools.make_test_intervals())
>>> tree
TimeIntervalTree([
  TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'}),
  TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
  TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
  TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'}),
  TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
  TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
  TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
  TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'}),
  TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
  TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'}),
  TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'}),
  TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])
```

**Example 1.** Explode intervals into optimal number of non-overlapping trees:

```
>>> exploded_trees = tree.explode_intervals()
>>> for exploded_tree in exploded_trees:
...     exploded_tree
...
TimeIntervalTree([
  TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'}),
  TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
  TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'})
])
TimeIntervalTree([
  TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
  TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
  TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'}),
  TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'}),
  TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'})
])
TimeIntervalTree([
  TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'}),
  TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'})
])
```



```

    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])

```

**Example 2.** Explode intervals into less-than-optimal number of overlapping trees:

```

>>> exploded_trees = tree.explode_intervals(aggregate_count=2)
>>> for exploded_tree in exploded_trees:
...     exploded_tree
...
TimeIntervalTree([
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'})
])
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'}),
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'}),
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])

```

**Example 3.** Explode intervals into greater-than-optimal number of non-overlapping trees:

```

>>> exploded_trees = tree.explode_intervals(aggregate_count=6)
>>> for exploded_tree in exploded_trees:
...     exploded_tree
...
TimeIntervalTree([
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'})
])
TimeIntervalTree([
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'})
])
TimeIntervalTree([
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'}),
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])
TimeIntervalTree([
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'})
])
TimeIntervalTree([
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'})
])
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'}),
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'})
])

```

Returns 0 or more trees.

`TimeIntervalTree.find_intervals_intersecting_or_tangent_to_interval(*args)`

Find all intervals in tree intersecting or tangent to the interval defined in *args*:

```

>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])

```

```

>>> interval = TimeInterval(0, 1)
>>> found = \
...     tree.find_intervals_intersecting_or_tangent_to_interval(

```

```
...     interval)
>>> sorted([x['name'] for x in found])
['a', 'b', 'c']
```

```
>>> interval = TimeInterval(3, 4)
>>> found = \
...     tree.find_intervals_intersecting_or_tangent_to_interval(
...         interval)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

Return *TimeIntervalTree* instance.

**TimeIntervalTree.find\_intervals\_intersecting\_or\_tangent\_to\_offset** (*offset*)

Find all intervals in tree intersecting or tangent to *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 1
>>> found = \
...     tree.find_intervals_intersecting_or_tangent_to_offset(
...         offset)
>>> sorted([x['name'] for x in found])
['a', 'b', 'c']
```

```
>>> offset = 3
>>> found = \
...     tree.find_intervals_intersecting_or_tangent_to_offset(
...         offset)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

Return *TimeIntervalTree* instance.

**TimeIntervalTree.find\_intervals\_starting\_after\_offset** (*offset*)

Find all intervals in tree starting after *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 0
>>> found = tree.find_intervals_starting_after_offset(offset)
>>> sorted([x['name'] for x in found])
['b', 'd']
```

```
>>> offset = 1
>>> found = tree.find_intervals_starting_after_offset(offset)
>>> sorted([x['name'] for x in found])
['d']
```

Return *TimeIntervalTree* instance.

**TimeIntervalTree.find\_intervals\_starting\_and\_stopping\_within\_interval** (*\*args*)

Find all intervals in tree starting and stopping within the interval defined by *args*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> interval = TimeInterval(1, 3)
>>> found = \
...     tree.find_intervals_starting_and_stopping_within_interval(
...         interval)
>>> sorted([x['name'] for x in found])
['b', 'd']
```

```
>>> interval = TimeInterval(-1, 2)
>>> found = \
...     tree.find_intervals_starting_and_stopping_within_interval(
...         interval)
>>> sorted([x['name'] for x in found])
['a', 'b']
```

Return *TimeIntervalTree* instance.

**TimeIntervalTree.find\_intervals\_starting\_at\_offset** (*offset*)

Find all intervals in tree starting at *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 0
>>> found = tree.find_intervals_starting_at_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'c']
```

```
>>> offset = 1
>>> found = tree.find_intervals_starting_at_offset(offset)
>>> sorted([x['name'] for x in found])
['b']
```

Return *TimeIntervalTree* instance.

**TimeIntervalTree.find\_intervals\_starting\_before\_offset** (*offset*)

Find all intervals in tree starting before *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 1
>>> found = tree.find_intervals_starting_before_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'c']
```

```
>>> offset = 2
>>> found = tree.find_intervals_starting_before_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b', 'c']
```

Return *TimeIntervalTree* instance.

**TimeIntervalTree.find\_intervals\_starting\_or\_stopping\_at\_offset** (*offset*)

Find all intervals in tree starting or stopping at *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 2
>>> found = \
...     tree.find_intervals_starting_or_stopping_at_offset(offset)
```

```
>>> sorted([x['name'] for x in found])
['b', 'd']
```

```
>>> offset = 1
>>> found = \
...     tree.find_intervals_starting_or_stopping_at_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.find_intervals_starting_within_interval(*args)`  
Find all intervals in tree starting within the interval defined by *args*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> interval = TimeInterval((-1, 2), (1, 2))
>>> found = tree.find_intervals_starting_within_interval(interval)
>>> sorted([x['name'] for x in found])
['a', 'c']
```

```
>>> interval = TimeInterval((1, 2), (5, 2))
>>> found = tree.find_intervals_starting_within_interval(interval)
>>> sorted([x['name'] for x in found])
['b', 'd']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.find_intervals_stopping_after_offset(offset)`  
Find all intervals in tree stopping after *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 1
>>> found = tree.find_intervals_stopping_after_offset(offset)
>>> sorted([x['name'] for x in found])
['b', 'c', 'd']
```

```
>>> offset = 2
>>> found = tree.find_intervals_stopping_after_offset(offset)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.find_intervals_stopping_at_offset(offset)`  
Find all intervals in tree stopping at *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 3
>>> found = tree.find_intervals_stopping_at_offset(offset)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

```
>>> offset = 1
>>> found = tree.find_intervals_stopping_at_offset(offset)
```

```
>>> sorted([x['name'] for x in found])
['a']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.find_intervals_stopping_before_offset` (*offset*)

Find all intervals in tree stopping before *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 3
>>> found = tree.find_intervals_stopping_before_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b']
```

```
>>> offset = (7, 2)
>>> found = tree.find_intervals_stopping_before_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b', 'c', 'd']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.find_intervals_stopping_within_interval` (*\*args*)

Find all intervals in tree stopping within the interval defined by *args*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> interval = TimeInterval((3, 2), (5, 2))
>>> found = tree.find_intervals_stopping_within_interval(interval)
>>> sorted([x['name'] for x in found])
['b']
```

```
>>> interval = TimeInterval((5, 2), (7, 2))
>>> found = tree.find_intervals_stopping_within_interval(interval)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.fuse_overlapping_intervals` (*include\_tangent\_intervals=False*)

Fuse overlapping intervals:

```
>>> a = TimeInterval(0, 10)
>>> b = TimeInterval(5, 15)
>>> c = TimeInterval(15, 25)
>>> tree = TimeIntervalTree([a, b, c])
>>> tree.fuse_overlapping_intervals()
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(15, 1), {}),
    TimeInterval(Offset(15, 1), Offset(25, 1), {})
])
```

```
>>> tree.fuse_overlapping_intervals(include_tangent_intervals=True)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(25, 1), {})
])
```

Returns time interval tree.

(*TimeIntervalMixin*).`get_overlap_with_interval` (*interval*)

Returns amount of overlap with *interval*.

`(TimeIntervalMixin).is_contained_by_interval(interval)`

True if interval is contained by *interval*.

`(TimeIntervalMixin).is_container_of_interval(interval)`

True if interval contains *interval*.

`(TimeIntervalMixin).is_overlapped_by_interval(interval)`

True if interval is overlapped by *interval*.

`(TimeIntervalMixin).is_tangent_to_interval(interval)`

True if interval is tangent to *interval*.

`(TimeIntervalAggregateMixin).partition(include_tangent_intervals=False)`

Partition aggregate into groups of overlapping intervals:

```
>>> tree = timeintervaltools.TimeIntervalTree(
...     timeintervaltools.make_test_intervals())
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'}),
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'}),
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'}),
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'}),
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'}),
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])
```

```
>>> for group in tree.partition():
...     group
...
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'})
])
TimeIntervalTree([
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'})
])
TimeIntervalTree([
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'})
])
TimeIntervalTree([
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'})
])
TimeIntervalTree([
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'})
])
TimeIntervalTree([
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])
```

If `include_tangent_intervals` is true, treat tangent intervals as part of the same group:

```
>>> for group in tree.partition(include_tangent_intervals=True):
...     group
...
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'})
])
TimeIntervalTree([
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
```

```

        TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'})
    ])
    TimeIntervalTree([
        TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
        TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
        TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
        TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'})
    ])
    TimeIntervalTree([
        TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
        TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'})
    ])
    TimeIntervalTree([
        TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'}),
        TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
    ])

```

Returns 0 or more trees.

`TimeIntervalTree.quantize_to_rational(rational)`

Quantize all intervals in tree to a multiple (1 or more) of *rational*:

```

>>> a = TimeInterval((1, 16), (1, 8), {'name': 'a'})
>>> b = TimeInterval((2, 7), (13, 7), {'name': 'b'})
>>> c = TimeInterval((3, 5), (8, 5), {'name': 'c'})
>>> d = TimeInterval((2, 3), (5, 3), {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(1, 16), Offset(1, 8), {'name': 'a'}),
    TimeInterval(Offset(2, 7), Offset(13, 7), {'name': 'b'}),
    TimeInterval(Offset(3, 5), Offset(8, 5), {'name': 'c'}),
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'd'})
])

```

```

>>> rational = (1, 4)
>>> tree.quantize_to_rational(rational)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 4), {'name': 'a'}),
    TimeInterval(Offset(1, 4), Offset(7, 4), {'name': 'b'}),
    TimeInterval(Offset(1, 2), Offset(3, 2), {'name': 'c'}),
    TimeInterval(Offset(3, 4), Offset(7, 4), {'name': 'd'})
])

```

```

>>> rational = (1, 3)
>>> tree.quantize_to_rational(rational)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 3), {'name': 'a'}),
    TimeInterval(Offset(1, 3), Offset(2, 1), {'name': 'b'}),
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'c'}),
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'd'})
])

```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.scale_by_rational(rational)`

Scale aggregate duration of tree by *rational*:

```

>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])

```

```

>>> result = tree.scale_by_rational((2, 3))
>>> result

```

```
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(2, 3), {'name': 'one'}),
    TimeInterval(Offset(1, 3), Offset(5, 3), {'name': 'two'}),
    TimeInterval(Offset(4, 3), Offset(8, 3), {'name': 'three'})
])
```

Scaling works regardless of the starting offset of the *TimeIntervalTree*:

```
>>> zero = TimeInterval(-4, 0, {'name': 'zero'})
>>> tree = TimeIntervalTree([zero, one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(-4, 1), Offset(0, 1), {'name': 'zero'}),
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.scale_by_rational(2)
>>> result
TimeIntervalTree([
    TimeInterval(Offset(-4, 1), Offset(4, 1), {'name': 'zero'}),
    TimeInterval(Offset(4, 1), Offset(6, 1), {'name': 'one'}),
    TimeInterval(Offset(5, 1), Offset(9, 1), {'name': 'two'}),
    TimeInterval(Offset(8, 1), Offset(12, 1), {'name': 'three'})
])
```

```
>>> result.start_offset == tree.start_offset
True
>>> result.duration == tree.duration * 2
True
```

Return *TimeIntervalTree* instance.

**TimeIntervalTree.scale\_interval\_durations\_by\_rational** (*rational*)  
Scale the duration of each interval by *rational*, maintaining their start\_offset offsets:

```
>>> a = timeintervaltools.TimeInterval(-1, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
>>> c = timeintervaltools.TimeInterval(9, 16)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> tree.scale_interval_durations_by_rational(Multiplier(6, 5))
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(19, 5), {}),
    TimeInterval(Offset(6, 1), Offset(66, 5), {}),
    TimeInterval(Offset(9, 1), Offset(87, 5), {})
])
```

Returns *TimeIntervalTree*.

**TimeIntervalTree.scale\_interval\_durations\_to\_rational** (*rational*)  
Scale the duration of each interval to *rational*, maintaining their start\_offset offsets:

```
>>> a = timeintervaltools.TimeInterval(-1, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
>>> c = timeintervaltools.TimeInterval(9, 16)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> tree.scale_interval_durations_to_rational(Duration(1, 7))
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(-6, 7), {}),
    TimeInterval(Offset(6, 1), Offset(43, 7), {}),
    TimeInterval(Offset(9, 1), Offset(64, 7), {})
])
```

Returns *TimeIntervalTree*.

**TimeIntervalTree.scale\_interval\_offsets\_by\_rational** (*rational*)  
Scale the starting offset of each interval by *rational*, maintaining the earliest startest offset:

```
>>> a = timeintervaltools.TimeInterval(-1, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
```



```
>>> c = timeintervaltools.TimeInterval(9, 16)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> tree.scale_interval_offsets_by_rational(Multiplier(4, 5))
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(3, 1), {}),
    TimeInterval(Offset(23, 5), Offset(53, 5), {}),
    TimeInterval(Offset(7, 1), Offset(14, 1), {})
])
```

Returns interval tree.

`TimeIntervalTree.scale_to_rational(rational)`

Scale aggregate duration of tree to *rational*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.scale_to_rational(1)
>>> result
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 4), {'name': 'one'}),
    TimeInterval(Offset(1, 8), Offset(5, 8), {'name': 'two'}),
    TimeInterval(Offset(1, 2), Offset(1, 1), {'name': 'three'})
])
```

```
>>> result.scale_to_rational(10)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(5, 2), {'name': 'one'}),
    TimeInterval(Offset(5, 4), Offset(25, 4), {'name': 'two'}),
    TimeInterval(Offset(5, 1), Offset(10, 1), {'name': 'three'})
])
```

Scaling works regardless of the starting offset of the *TimeIntervalTree*:

```
>>> zero = TimeInterval(-4, 0, {'name': 'zero'})
>>> tree = TimeIntervalTree([zero, one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(-4, 1), Offset(0, 1), {'name': 'zero'}),
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> tree.scale_to_rational(4)
TimeIntervalTree([
    TimeInterval(Offset(-4, 1), Offset(-2, 1), {'name': 'zero'}),
    TimeInterval(Offset(-2, 1), Offset(-3, 2), {'name': 'one'}),
    TimeInterval(Offset(-7, 4), Offset(-3, 4), {'name': 'two'}),
    TimeInterval(Offset(-1, 1), Offset(0, 1), {'name': 'three'})
])
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.shift_by_rational(rational)`

Shift aggregate offset of tree by *rational*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
```

```
TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.shift_by_rational(-2.5)
>>> result
TimeIntervalTree([
    TimeInterval(Offset(-5, 2), Offset(-3, 2), {'name': 'one'}),
    TimeInterval(Offset(-2, 1), Offset(0, 1), {'name': 'two'}),
    TimeInterval(Offset(-1, 2), Offset(3, 2), {'name': 'three'})
])
>>> result.shift_by_rational(6)
TimeIntervalTree([
    TimeInterval(Offset(7, 2), Offset(9, 2), {'name': 'one'}),
    TimeInterval(Offset(4, 1), Offset(6, 1), {'name': 'two'}),
    TimeInterval(Offset(11, 2), Offset(15, 2), {'name': 'three'})
])
```

Return *TimeIntervalTree* instance.

*TimeIntervalTree*.**shift\_to\_rational** (*rational*)

Shift aggregate offset of tree to *rational*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.shift_to_rational(100)
>>> result
TimeIntervalTree([
    TimeInterval(Offset(100, 1), Offset(101, 1), {'name': 'one'}),
    TimeInterval(Offset(201, 2), Offset(205, 2), {'name': 'two'}),
    TimeInterval(Offset(102, 1), Offset(104, 1), {'name': 'three'})
])
```

Return *TimeIntervalTree* instance.

*TimeIntervalTree*.**split\_at\_rationals** (*\*rationals*)

Split tree at each rational in *rationals*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.split_at_rationals(1, 2, 3)
>>> len(result)
4
```

```
>>> result[0]
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(1, 1), {'name': 'two'})
])
```

```
>>> result[1]
TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'two'})
])
```

```
>>> result[2]
TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'three'})
])
```

```
>>> result[3]
TimeIntervalTree([
    TimeInterval(Offset(3, 1), Offset(4, 1), {'name': 'three'})
])
```

Returns tuple of *TimeIntervalTree* instances.

## Special methods

```
TimeIntervalTree.__contains__(item)
TimeIntervalTree.__copy__()
TimeIntervalTree.__eq__(expr)
TimeIntervalTree.__getitem__(item)
TimeIntervalTree.__getslice__(start_offset, end)
TimeIntervalTree.__iter__()
TimeIntervalTree.__len__()
TimeIntervalTree.__ne__(expr)
TimeIntervalTree.__nonzero__()
```

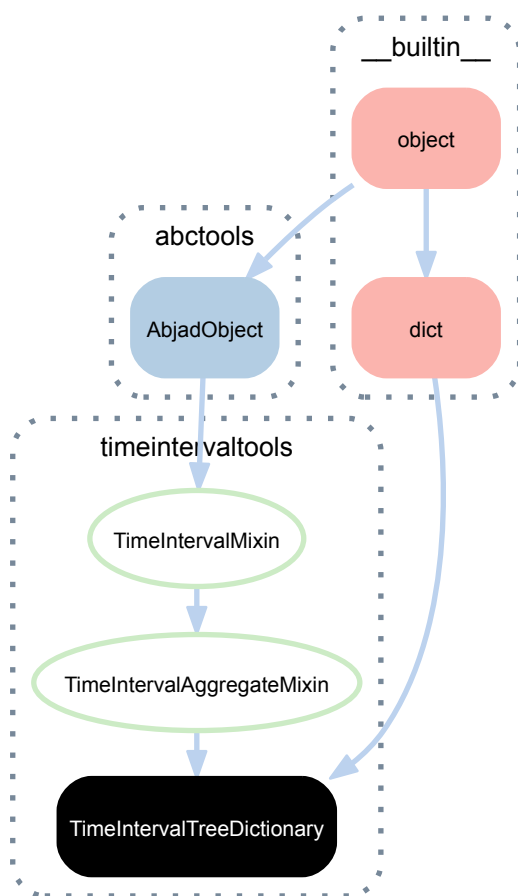
*TimeIntervalTree* evaluates to True if it contains any intervals:

```
>>> true_tree = TimeIntervalTree([TimeInterval(0, 1)])
>>> false_tree = TimeIntervalTree([])
```

```
>>> bool(true_tree)
True
>>> bool(false_tree)
False
```

Returns boolean.

```
TimeIntervalTree.__repr__()
```

36.2.3 `timeintervaltools.TimeIntervalTreeDictionary`

**class** `timeintervaltools.TimeIntervalTreeDictionary(*args)`

A dictionary of time-interval trees.

```

>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})

```

*TimeIntervalTreeDictionary* can be instantiated from one or more other *TimeIntervalTreeDictionary* instances, whose trees will be fused if they share keys. It can also be instantiated from a regular dictionary whose values are *TimeIntervalTree* instances, or from a list of pairs where the second value of each pair is a *TimeIntervalTree* instance.

*TimeIntervalTreeDictionary* supports the same set of methods and properties as *TimeIntervalTree* and *TimeInterval*, including searching for intervals, quantizing, scaling, shifting and splitting.

*TimeIntervalTreeDictionary* is immutable.

Return *TimeIntervalTreeDictionary* instance.

## Bases

- `__builtin__.dict`
- `timeintervaltools.TimeIntervalAggregateMixin`
- `timeintervaltools.TimeIntervalMixin`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(*TimeIntervalAggregateMixin*).**all\_unique\_bounds**

(*TimeIntervalMixin*).**bounds**

Start and stop\_offset of self returned as *TimeInterval* instance:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

(*TimeIntervalMixin*).**center**

Center offset of start\_offset and stop\_offset offsets:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.center
Offset(6, 1)
```

Returns *Offset* instance.

*TimeIntervalTreeDictionary*.**composite\_tree**

The *TimeIntervalTree* composed of all the intervals in all trees in self:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.composite_tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})
])
```

Return *TimeIntervalTree* instance.

(TimeIntervalMixin).**duration**

Duration of the time interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.duration
Duration(8, 1)
```

Returns *Duration* instance.

TimeIntervalTreeDictionary.**earliest\_start**

The earliest start\_offset offset of all intervals in all trees in self:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.earliest_start
Offset(0, 1)
```

Return *Offset* instance.

TimeIntervalTreeDictionary.**earliest\_stop**

The earliest stop\_offset offset of all intervals in all trees in self:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.earliest_stop
Offset(1, 1)
```

Return *Offset* instance.

TimeIntervalTreeDictionary.**intervals**

TimeIntervalTreeDictionary.**latest\_start**

The latest start\_offset offset of all intervals in all trees in self:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.latest_start
Offset(2, 1)
```

Return *Offset* instance.

TimeIntervalTreeDictionary.**latest\_stop**

The latest stop\_offset offset of all intervals in all trees in self:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.latest_stop
Offset(3, 1)
```

Return *Offset* instance.

(TimeIntervalAggregateMixin).**offset\_counts**

(TimeIntervalAggregateMixin).**offsets**

(TimeIntervalMixin).**signature**

Tuple of start\_offset bound and stop\_offset bound.

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.signature
(Offset(2, 1), Offset(10, 1))
```

Returns 2-tuple of *Offset* instances.

(TimeIntervalMixin).**start\_offset**

Starting offset of interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.start_offset
Offset(2, 1)
```

Returns *Offset* instance.

(TimeIntervalMixin).**stop\_offset**

Stopping offset of interval:

```
>>> interval = timeintervaltools.TimeInterval(2, 10)
>>> interval.stop_offset
Offset(10, 1)
```

Returns *Offset* instance.

(TimeIntervalAggregateMixin).**storage\_format**

Storage format of time interval aggregate mixin.

Returns string.

## Methods

(TimeIntervalAggregateMixin).**calculate\_attack\_density** (*bounding\_interval=None*)

(TimeIntervalAggregateMixin).**calculate\_depth\_centroid** (*bounding\_interval=None*)

Calculate the weighted mean offset of *intervals*, such that the centroids of each interval in the depth tree of *intervals* make up the values of the mean, and the depth of each interval in the depth tree of *intervals* make up the weights.

Returns *Offset*.

(TimeIntervalAggregateMixin).**calculate\_depth\_density** (*bounding\_interval=None*)

Returns a fraction, of the duration of each interval in the depth tree of *intervals*, multiplied by the depth at that interval, divided by the overall duration of *intervals*.

The depth density of a single interval is *l*:

Returns multiplier.

`(TimeIntervalAggregateMixin).calculate_mean_attack_offset()`

`(TimeIntervalAggregateMixin).calculate_mean_release_offset()`

`(TimeIntervalAggregateMixin).calculate_minimum_mean_and_maximum_depths()`

Returns a 3-tuple of the minimum, mean and maximum depth of *intervals*.

If *intervals* is empty, return None.

“Mean” in this case is a weighted mean, where the durations of the intervals in depth tree of *intervals* are the weights.

`(TimeIntervalAggregateMixin).calculate_minimum_mean_and_maximum_durations()`

Returns a 3-tuple of the minimum, mean and maximum duration of all intervals in *intervals*.

If *intervals* is empty, return None.

`(TimeIntervalAggregateMixin).calculate_release_density(bounding_interval=None)`

`(TimeIntervalAggregateMixin).calculate_sustain_centroid()`

Returns a weighted mean, such that the centroid of each interval in *intervals* are the values, and the weights are their durations.

`(dict).clear()` → None. Remove all items from D.

`TimeIntervalTreeDictionary.clip_interval_durations_to_range(minimum=None, maximum=None)`

`(TimeIntervalAggregateMixin).compute_depth(bounding_interval=None)`

Compute a tree whose intervals represent the level of overlap of the time interval aggregate:

```
>>> a = timeintervaltools.TimeInterval(0, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
>>> c = timeintervaltools.TimeInterval(9, 15)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> tree.compute_depth()
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'depth': 1}),
    TimeInterval(Offset(3, 1), Offset(6, 1), {'depth': 0}),
    TimeInterval(Offset(6, 1), Offset(9, 1), {'depth': 1}),
    TimeInterval(Offset(9, 1), Offset(12, 1), {'depth': 2}),
    TimeInterval(Offset(12, 1), Offset(15, 1), {'depth': 1})
])
```

If *bounding\_interval* is not none, only consider the depth of time intervals which intersect that time interval:

```
>>> a = timeintervaltools.TimeInterval(0, 3)
>>> b = timeintervaltools.TimeInterval(6, 12)
>>> c = timeintervaltools.TimeInterval(9, 15)
>>> tree = timeintervaltools.TimeIntervalTree([a, b, c])
>>> d = timeintervaltools.TimeInterval(-1, 16)
>>> tree.compute_depth(bounding_interval=d)
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(0, 1), {'depth': 0}),
    TimeInterval(Offset(0, 1), Offset(3, 1), {'depth': 1}),
    TimeInterval(Offset(3, 1), Offset(6, 1), {'depth': 0}),
    TimeInterval(Offset(6, 1), Offset(9, 1), {'depth': 1}),
    TimeInterval(Offset(9, 1), Offset(12, 1), {'depth': 2}),
    TimeInterval(Offset(12, 1), Offset(15, 1), {'depth': 1}),
    TimeInterval(Offset(15, 1), Offset(16, 1), {'depth': 0})
])
```

Returns interval tree.

`(TimeIntervalAggregateMixin).compute_logical_and(bounding_interval=None)`

Compute logical AND of intervals.

Returns time interval tree.



`(TimeIntervalAggregateMixin).compute_logical_not (bounding_interval=None)`  
 Compute logical NOT of intervals.

Returns time interval tree.

`(TimeIntervalAggregateMixin).compute_logical_or (bounding_interval=None)`  
 Compute logical OR of intervals.

Returns time interval tree.

`(TimeIntervalAggregateMixin).compute_logical_xor (bounding_interval=None)`  
 Compute logical XOR of intervals.

Returns time interval tree.

`(dict).copy()` → a shallow copy of D

`TimeIntervalTreeDictionary.find_intervals_intersecting_or_tangent_to_interval (*args)`  
 Find all intervals in dictionary intersecting or tangent to the interval defined in *args*:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})
```

```
>>> interval = timeintervaltools.TimeInterval(0, 1)
>>> treedict.find_intervals_intersecting_or_tangent_to_interval(
...     interval)
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
})
```

```
>>> interval = timeintervaltools.TimeInterval(3, 4)
>>> treedict.find_intervals_intersecting_or_tangent_to_interval(
...     interval)
TimeIntervalTreeDictionary({
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.find_intervals_intersecting_or_tangent_to_offset` (*offset*)  
Find all intervals in dictionary intersecting or tangent to *offset*:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_intersecting_or_tangent_to_offset(
...     offset)
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
})
```

```
>>> offset = 3
>>> treedict.find_intervals_intersecting_or_tangent_to_offset(
...     offset)
TimeIntervalTreeDictionary({
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.find_intervals_starting_after_offset` (*offset*)  
Find all intervals in dictionary starting after *offset*:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
```

```

...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})

```

```

>>> offset = 0
>>> treedict.find_intervals_starting_after_offset(offset)
TimeIntervalTreeDictionary({
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})

```

```

>>> offset = 1
>>> treedict.find_intervals_starting_after_offset(offset)
TimeIntervalTreeDictionary({
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})

```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**find\_intervals\_starting\_and\_stopping\_within\_interval** (\*args)  
Find all intervals in dictionary starting and stopping within the interval defined by args:

```

>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})

```

```

>>> interval = timeintervaltools.TimeInterval(1, 3)
>>> treedict.find_intervals_starting_and_stopping_within_interval(
...     interval)
TimeIntervalTreeDictionary({
  'b': TimeIntervalTree([

```

```

        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})

```

```

>>> interval = timeintervaltools.TimeInterval(-1, 2)
>>> treedict.find_intervals_starting_and_stopping_within_interval(
...     interval)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
})

```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**find\_intervals\_starting\_at\_offset** (*offset*)

Find all intervals in dictionary starting at *offset*:

```

>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})

```

```

>>> offset = 0
>>> treedict.find_intervals_starting_at_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
})

```

```

>>> offset = 1
>>> treedict.find_intervals_starting_at_offset(offset)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
})

```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**find\_intervals\_starting\_before\_offset** (*offset*)

Find all intervals in dictionary starting before *offset*:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_starting_before_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
})
```

```
>>> offset = 2
>>> treedict.find_intervals_starting_before_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**find\_intervals\_starting\_or\_stopping\_at\_offset** (*offset*)

Find all intervals in dictionary starting or stopping at *offset*:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 2
>>> treedict.find_intervals_starting_or_stopping_at_offset(offset)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_starting_or_stopping_at_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**find\_intervals\_starting\_within\_interval**(\*args)  
Find all intervals in dictionary starting within the interval defined by *args*:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> interval = timeintervaltools.TimeInterval((-1, 2), (1, 2))
>>> treedict.find_intervals_starting_within_interval(interval)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'c': TimeIntervalTree([
```

```

        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'})),
    ]),
})

```

```

>>> interval = timeintervaltools.TimeInterval((1, 2), (5, 2))
>>> treedict.find_intervals_starting_within_interval(interval)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'})),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})),
    ]),
})

```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**find\_intervals\_stopping\_after\_offset** (*offset*)  
Find all intervals in dictionary stopping after *offset*:

```

>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'})),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'})),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'})),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})),
    ]),
})

```

```

>>> offset = 1
>>> treedict.find_intervals_stopping_after_offset(offset)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'})),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'})),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})),
    ]),
})

```

```

>>> offset = 2
>>> treedict.find_intervals_stopping_after_offset(offset)
TimeIntervalTreeDictionary({
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'})),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})),
    ]),
})

```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.find_intervals_stopping_at_offset` (*offset*)  
Find all intervals in dictionary stopping at *offset*:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})
```

```
>>> offset = 3
>>> treedict.find_intervals_stopping_at_offset(offset)
TimeIntervalTreeDictionary({
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_stopping_at_offset(offset)
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.find_intervals_stopping_before_offset` (*offset*)  
Find all intervals in dictionary stopping before *offset*:

```
>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
```



```

        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'})),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})),
    ]),
})

```

```

>>> offset = 3
>>> treedict.find_intervals_stopping_before_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'})),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'})),
    ]),
})

```

```

>>> offset = (7, 2)
>>> treedict.find_intervals_stopping_before_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'})),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'})),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'})),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})),
    ]),
})

```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.find_intervals_stopping_within_interval(*args)`  
Find all intervals in dictionary stopping within the interval defined by *args*:

```

>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(1, 2, {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 3, {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'})),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'})),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'})),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})),
    ]),
})

```

```

>>> interval = timeintervaltools.TimeInterval((3, 2), (5, 2))
>>> treedict.find_intervals_stopping_within_interval(interval)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'})),
    ]),
})

```

```
    ]),
  })
```

```
>>> interval = timeintervaltools.TimeInterval((5, 2), (7, 2))
>>> treedict.find_intervals_stopping_within_interval(interval)
TimeIntervalTreeDictionary({
  'c': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
  ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**fuse\_overlapping\_intervals** (*include\_tangent\_intervals=False*)

(dict).**.get** (*k*, *d*) → D[k] if k in D, else d. d defaults to None.

(*TimeIntervalMixin*).**.get\_overlap\_with\_interval** (*interval*)  
Returns amount of overlap with *interval*.

(dict).**.has\_key** (*k*) → True if D has a key k, else False

(*TimeIntervalMixin*).**.is\_contained\_by\_interval** (*interval*)  
True if interval is contained by *interval*.

(*TimeIntervalMixin*).**.is\_container\_of\_interval** (*interval*)  
True if interval contains *interval*.

(*TimeIntervalMixin*).**.is\_overlapped\_by\_interval** (*interval*)  
True if interval is overlapped by *interval*.

(*TimeIntervalMixin*).**.is\_tangent\_to\_interval** (*interval*)  
True if interval is tangent to *interval*.

(dict).**.items** () → list of D's (key, value) pairs, as 2-tuples

(dict).**.iteritems** () → an iterator over the (key, value) items of D

(dict).**.iterkeys** () → an iterator over the keys of D

(dict).**.itervalues** () → an iterator over the values of D

(dict).**.keys** () → list of D's keys

(*TimeIntervalAggregateMixin*).**.partition** (*include\_tangent\_intervals=False*)  
Partition aggregate into groups of overlapping intervals:

```
>>> tree = timeintervaltools.TimeIntervalTree(
...     timeintervaltools.make_test_intervals())
>>> tree
TimeIntervalTree([
  TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'}),
  TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
  TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
  TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'}),
  TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
  TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
  TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
  TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'}),
  TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
  TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'}),
  TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'}),
  TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])
```

```
>>> for group in tree.partition():
...     group
...
TimeIntervalTree([
```

```

    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'})
])
TimeIntervalTree([
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'})
])
TimeIntervalTree([
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'})
])
TimeIntervalTree([
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'})
])
TimeIntervalTree([
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'})
])
TimeIntervalTree([
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])

```

If `include_tangent_intervals` is true, treat tangent intervals as part of the same group:

```

>>> for group in tree.partition(include_tangent_intervals=True):
...     group
...
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'a'})
])
TimeIntervalTree([
    TimeInterval(Offset(5, 1), Offset(13, 1), {'name': 'b'}),
    TimeInterval(Offset(6, 1), Offset(10, 1), {'name': 'c'}),
    TimeInterval(Offset(8, 1), Offset(9, 1), {'name': 'd'})
])
TimeIntervalTree([
    TimeInterval(Offset(15, 1), Offset(23, 1), {'name': 'e'}),
    TimeInterval(Offset(16, 1), Offset(21, 1), {'name': 'f'}),
    TimeInterval(Offset(17, 1), Offset(19, 1), {'name': 'g'}),
    TimeInterval(Offset(19, 1), Offset(20, 1), {'name': 'h'})
])
TimeIntervalTree([
    TimeInterval(Offset(25, 1), Offset(30, 1), {'name': 'i'}),
    TimeInterval(Offset(26, 1), Offset(29, 1), {'name': 'j'})
])
TimeIntervalTree([
    TimeInterval(Offset(32, 1), Offset(34, 1), {'name': 'k'}),
    TimeInterval(Offset(34, 1), Offset(37, 1), {'name': 'l'})
])

```

Returns 0 or more trees.

(dict) **.pop**(*k*, *d*) → *v*, remove specified key and return the corresponding value.

If key is not found, *d* is returned if given, otherwise `KeyError` is raised

(dict) **.popitem**() → (*k*, *v*), remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if *D* is empty.

`TimeIntervalTreeDictionary.quantize_to_rational(rational)`

Quantize all intervals in dictionary to a multiple (1 or more) of *rational*:

```

>>> a = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((1, 16), (1, 8), {'name': 'a'})])
>>> b = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((2, 7), (13, 7), {'name': 'b'})])
>>> c = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((3, 5), (8, 5), {'name': 'c'})])
>>> d = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((2, 3), (5, 3), {'name': 'd'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(

```

```

...     {'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(1, 16), Offset(1, 8), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(2, 7), Offset(13, 7), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(3, 5), Offset(8, 5), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'd'}),
  ]),
})

```

```

>>> rational = (1, 4)
>>> treedict.quantize_to_rational(rational)
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 4), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 4), Offset(7, 4), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(1, 2), Offset(3, 2), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(3, 4), Offset(7, 4), {'name': 'd'}),
  ]),
})

```

```

>>> rational = (1, 3)
>>> treedict.quantize_to_rational(rational)
TimeIntervalTreeDictionary({
  'a': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 3), {'name': 'a'}),
  ]),
  'b': TimeIntervalTree([
    TimeInterval(Offset(1, 3), Offset(2, 1), {'name': 'b'}),
  ]),
  'c': TimeIntervalTree([
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'c'}),
  ]),
  'd': TimeIntervalTree([
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'd'}),
  ]),
})

```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**scale\_by\_rational** (*rational*)

Scale aggregate duration of dictionary by *rational*:

```

>>> one = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'one'})])
>>> two = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
  'one': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
  ]),
  'three': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
  ]),
})

```

```

        'two': TimeIntervalTree([
            TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
        ]),
    })

```

```

>>> result = treedict.scale_by_rational((2, 3))
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(2, 3), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(4, 3), Offset(8, 3), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 3), Offset(5, 3), {'name': 'two'}),
    ]),
})

```

Scaling works regardless of the starting offset of the *TimeIntervalTreeDictionary*:

```

>>> zero = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(-4, 0, {'name': 'zero'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'zero': zero, 'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
    'zero': TimeIntervalTree([
        TimeInterval(Offset(-4, 1), Offset(0, 1), {'name': 'zero'}),
    ]),
})

```

```

>>> result = treedict.scale_by_rational(2)
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(4, 1), Offset(6, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(8, 1), Offset(12, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(5, 1), Offset(9, 1), {'name': 'two'}),
    ]),
    'zero': TimeIntervalTree([
        TimeInterval(Offset(-4, 1), Offset(4, 1), {'name': 'zero'}),
    ]),
})

```

```

>>> result.start_offset == treedict.start_offset
True
>>> result.duration == treedict.duration * 2
True

```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**scale\_interval\_durations\_by\_rational** (*rational*)

*TimeIntervalTreeDictionary*.**scale\_interval\_durations\_to\_rational** (*rational*)

*TimeIntervalTreeDictionary*.**scale\_interval\_offsets\_by\_rational** (*rational*)

`TimeIntervalTreeDictionary.scale_to_rational` (*rational*)

Scale aggregate duration of dictionary to *rational*:

```
>>> one = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'one'})])
>>> two = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
})
```

```
>>> result = treedict.scale_to_rational(1)
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 4), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(1, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 8), Offset(5, 8), {'name': 'two'}),
    ]),
})
```

```
>>> result.scale_to_rational(10)
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(5, 2), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(5, 1), Offset(10, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(5, 4), Offset(25, 4), {'name': 'two'}),
    ]),
})
```

Scaling works regardless of the starting offset of the *TimeIntervalTreeDictionary*:

```
>>> zero = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(-4, 0, {'name': 'zero'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'zero': zero, 'one': one, 'two': two, 'three': three})
```

```
>>> treedict.scale_to_rational(4)
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(-2, 1), Offset(-3, 2), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(-1, 1), Offset(0, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(-7, 4), Offset(-3, 4), {'name': 'two'}),
    ]),
    'zero': TimeIntervalTree([
        TimeInterval(Offset(-4, 1), Offset(-2, 1), {'name': 'zero'}),
    ]),
})
```

```
})
```

Return *TimeIntervalTreeDictionary* instance.

(dict) **.setdefault** (*k*, *d*) → D.get(*k*,*d*), also set D[*k*]=*d* if *k* not in D

*TimeIntervalTreeDictionary*.**shift\_by\_rational** (*rational*)

Shift aggregate offset of dictionary by *rational*:

```
>>> one = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'one'})])
>>> two = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
})
```

```
>>> result = treedict.shift_by_rational(-2.5)
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(-5, 2), Offset(-3, 2), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(-1, 2), Offset(3, 2), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(-2, 1), Offset(0, 1), {'name': 'two'}),
    ]),
})
```

```
>>> result.shift_by_rational(6)
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(7, 2), Offset(9, 2), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(11, 2), Offset(15, 2), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(4, 1), Offset(6, 1), {'name': 'two'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**shift\_to\_rational** (*rational*)

Shift aggregate offset of dictionary to *rational*:

```
>>> one = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'one'})])
>>> two = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
```

```
'one': TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
]),
'three': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
]),
'two': TimeIntervalTree([
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
]),
})
```

```
>>> result = treedict.shift_to_rational(100)
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(100, 1), Offset(101, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(102, 1), Offset(104, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(201, 2), Offset(205, 2), {'name': 'two'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

*TimeIntervalTreeDictionary*.**split\_at\_rationals**(\*rationals)

Split dictionary at each rational in *rationals*:

```
>>> one = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(0, 1, {'name': 'one'})])
>>> two = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = timeintervaltools.TimeIntervalTree(
...     [timeintervaltools.TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = timeintervaltools.TimeIntervalTreeDictionary(
...     {'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
})
```

```
>>> result = treedict.split_at_rationals(1, 2, 3)
>>> len(result)
4
```

```
>>> result[0]
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(1, 1), {'name': 'two'}),
    ]),
})
```

```
>>> result[1]
TimeIntervalTreeDictionary({
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'two'}),
    ]),
})
```



```
>>> result[2]
TimeIntervalTreeDictionary({
  'three': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'three'}),
  ]),
  'two': TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(5, 2), {'name': 'two'}),
  ]),
})
```

```
>>> result[3]
TimeIntervalTreeDictionary({
  'three': TimeIntervalTree([
    TimeInterval(Offset(3, 1), Offset(4, 1), {'name': 'three'}),
  ]),
})
```

Returns tuple of *TimeIntervalTreeDictionary* instances.

(dict).**update**(*[E]*, *\*\*F*) → None. Update D from dict/iterable E and F.  
 If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method,  
 does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

(dict).**values**() → list of D's values

(dict).**viewitems**() → a set-like object providing a view on D's items

(dict).**viewkeys**() → a set-like object providing a view on D's keys

(dict).**viewvalues**() → an object providing a view on D's values

## Special methods

(dict).**\_\_cmp\_\_**(y) <==> *cmp(x, y)*

(dict).**\_\_contains\_\_**(k) → True if D has a key k, else False

(dict).**\_\_delitem\_\_**()  
 x.**\_\_delitem\_\_**(y) <==> del x[y]

(dict).**\_\_eq\_\_**()  
 x.**\_\_eq\_\_**(y) <==> x==y

(dict).**\_\_ge\_\_**()  
 x.**\_\_ge\_\_**(y) <==> x>=y

(dict).**\_\_getitem\_\_**()  
 x.**\_\_getitem\_\_**(y) <==> x[y]

(dict).**\_\_gt\_\_**()  
 x.**\_\_gt\_\_**(y) <==> x>y

(dict).**\_\_iter\_\_**() <==> *iter(x)*

(dict).**\_\_le\_\_**()  
 x.**\_\_le\_\_**(y) <==> x<=y

(dict).**\_\_len\_\_**() <==> *len(x)*

(dict).**\_\_lt\_\_**()  
 x.**\_\_lt\_\_**(y) <==> x<y

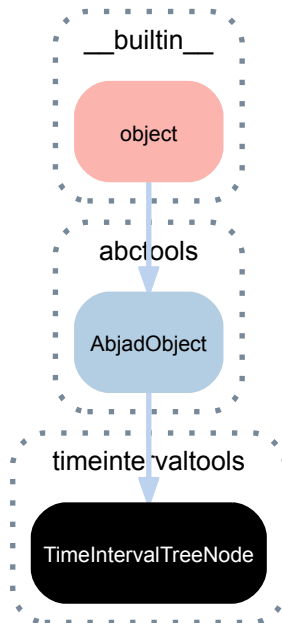
(dict).**\_\_ne\_\_**()  
 x.**\_\_ne\_\_**(y) <==> x!=y

(TimeIntervalMixin).**\_\_nonzero\_\_**()

TimeIntervalTreeDictionary.**\_\_repr\_\_**()

```
(dict).__setitem__()  
x.__setitem__(i, y) <==> x[i]=y
```

### 36.2.4 timeintervaltools.TimeIntervalTreeNode



**class** timeintervaltools.**TimeIntervalTreeNode** (*key*, *intervals=None*)  
 A red-black node in an TimeIntervalTree.

Duplicate payloads are supported by maintaining a list of TimeIntervals.

Not composer-safe.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`TimeIntervalTreeNode.grandparent`  
`TimeIntervalTreeNode.is_left_child`  
`TimeIntervalTreeNode.is_right_child`  
`TimeIntervalTreeNode.key`  
`TimeIntervalTreeNode.payload`  
`TimeIntervalTreeNode.sibling`  
`TimeIntervalTreeNode.uncle`

#### Read/write properties

`TimeIntervalTreeNode.earliest_stop`  
`TimeIntervalTreeNode.latest_stop`

TimeIntervalTreeNode.**left**  
 TimeIntervalTreeNode.**parent**  
 TimeIntervalTreeNode.**red**  
 TimeIntervalTreeNode.**right**

### Special methods

(AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

## 36.3 Functions

### 36.3.1 timeintervaltools.concatenate\_trees

timeintervaltools.**concatenate\_trees**(*trees*, *padding=0*)  
 Merge all trees in *trees*, offsetting each subsequent tree to start\_offset after the previous.  
 Returns TimeIntervalTree.

### 36.3.2 timeintervaltools.make\_test\_intervals

timeintervaltools.**make\_test\_intervals**()

### 36.3.3 timeintervaltools.mask\_intervals\_with\_intervals

timeintervaltools.**mask\_intervals\_with\_intervals**(*masked\_intervals*,  
*mask\_intervals*)  
 Clip or remove all intervals in *masked\_intervals* outside of the bounds defined in *mask\_intervals*, while maintaining *masked\_intervals*' payload contents:

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree

>>> a = TimeInterval(0, 10, {'a': 1})
>>> b = TimeInterval(5, 15, {'b': 2})
>>> tree = TimeIntervalTree([a, b])
>>> mask = TimeInterval(4, 11)
>>> timeintervaltools.mask_intervals_with_intervals(tree, mask)
TimeIntervalTree([
  TimeInterval(Offset(4, 1), Offset(10, 1), {'a': 1}),
  TimeInterval(Offset(5, 1), Offset(11, 1), {'b': 2})
])
```

Returns TimeIntervalTree.

### 36.3.4 `timeintervaltools.resolve_overlaps_between_nonoverlapping_trees`

`timeintervaltools.resolve_overlaps_between_nonoverlapping_trees` (*trees*,  
*mini-*  
*mum\_duration=None*)

Create a nonoverlapping `TimeIntervalTree` from *trees*.

Intervals in higher-indexed trees in *trees* only appear in part or whole where they do not overlap intervals from starter-indexed trees:

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree

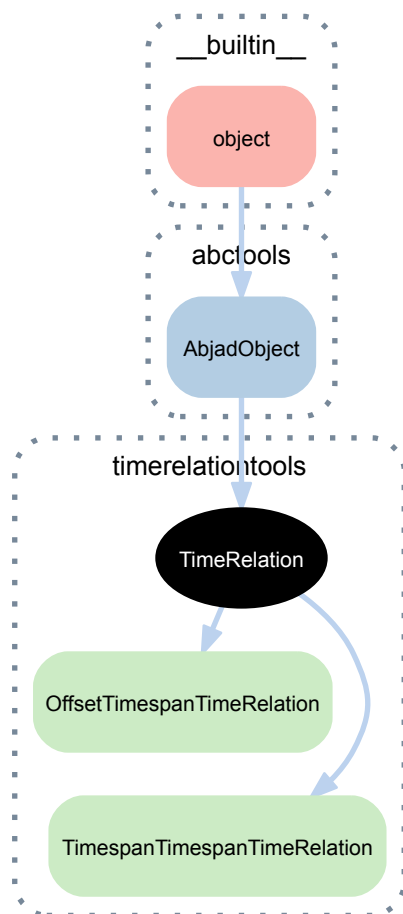
>>> a = TimeIntervalTree(TimeInterval(0, 4, {'a': 1}))
>>> b = TimeIntervalTree(TimeInterval(1, 5, {'b': 2}))
>>> c = TimeIntervalTree(TimeInterval(2, 6, {'c': 3}))
>>> d = TimeIntervalTree(TimeInterval(1, 3, {'d': 4}))
>>> timeintervaltools.resolve_overlaps_between_nonoverlapping_trees([a, b, c, d])
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(4, 1), {'a': 1}),
    TimeInterval(Offset(4, 1), Offset(5, 1), {'b': 2}),
    TimeInterval(Offset(5, 1), Offset(6, 1), {'c': 3})
])
```

Returns `TimeIntervalTree`.

# TIMERELATIONTOOLS

## 37.1 Abstract classes

### 37.1.1 timerelationtools.TimeRelation



**class** `timerelationtools.TimeRelation` (*inequality*)  
A time relation.

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1,
...     timespan_2=timespan_2,
...     hold=True,
...     )
```

```
>>> print time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
  ],
  logical_operator='and'
),
  timespan_1=timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(10, 1)
  ),
  timespan_2=timespantools.Timespan(
    start_offset=durationtools.Offset(5, 1),
    stop_offset=durationtools.Offset(15, 1)
  )
)
```

Time relations are immutable.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`TimeRelation.inequality`

Time relation inequality.

`TimeRelation.is_fully_loaded`

True when both time relation terms are not none. Otherwise false:

```
>>> time_relation.is_fully_loaded
True
```

Returns boolean.

`TimeRelation.is_fully_unloaded`

True when both time relation terms are none. Otherwise false:

```
>>> time_relation.is_fully_unloaded
False
```

Returns boolean.

`TimeRelation.storage_format`

Time relation storage format:

```
>>> print time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
  ],
  logical_operator='and'
),
  timespan_1=timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(10, 1)
  ),
  timespan_2=timespantools.Timespan(
    start_offset=durationtools.Offset(5, 1),
    stop_offset=durationtools.Offset(15, 1)
  )
)
```

Returns string.

## Methods

`TimeRelation.new(**kwargs)`

Initialize new time relation with keyword arguments optionally changed:

```
>>> time_relation = \
...     timerelationtools.timespan_2_stops_when_timespan_1_starts()
>>> new_time_relation = \
...     time_relation.new(timespan_1=timespantools.Timespan(0, 5))

>>> print time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
  ],
    logical_operator='and'
  )
)

>>> print new_time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
  ],
    logical_operator='and'
  ),
  timespan_1=timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(5, 1)
  )
)
```

Returns newly constructed time relation.

## Special methods

`TimeRelation.__call__()`

Evaluate time relation:

```
>>> time_relation()
True
```

Returns boolean.

`TimeRelation.__eq__(expr)`

True when *expr* is a equal-valued time relation. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

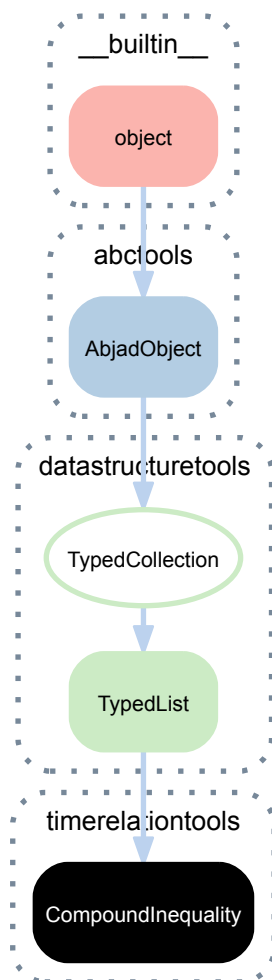
`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 37.2 Concrete classes

### 37.2.1 timerelationtools.CompoundInequality



**class** timerelationtools.**CompoundInequality** (*tokens=None*, *logical\_operator='and'*, *name=None*)

A compound time-relation inequality.

```
>>> compound_inequality = timerelationtools.CompoundInequality([
...     timerelationtools.CompoundInequality([
...         'timespan_1.start_offset <= timespan_2.start_offset',
...         'timespan_2.start_offset < timespan_1.stop_offset'],
...         logical_operator='and'),
...     timerelationtools.CompoundInequality([
...         'timespan_2.start_offset <= timespan_1.start_offset',
...         'timespan_1.start_offset < timespan_2.stop_offset'],
...         logical_operator='and')],
...     logical_operator='or',
... )
```

```
>>> print compound_inequality.storage_format
timerelationtools.CompoundInequality([
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
    ],
        logical_operator='and'
    ),
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
    ],
        logical_operator='and'
    )
])
```



```

        ],
        logical_operator='and'
    )
],
logical_operator='or'
)

```

## Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`CompoundInequality.logical_operator`  
Compound inequality logical operator.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedList).append(token)`  
Change *token* to item and append:

```

>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]

```

```

>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]

```

Returns none.

`(TypedList).count(token)`  
Change *token* to item and return count.

```

>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]

```

```

>>> integer_collection.count(0)
3

```

Returns count.

`CompoundInequality.evaluate(timespan_1_start_offset, timespan_1_stop_offset, timespan_2_start_offset, timespan_2_stop_offset)`

`CompoundInequality.evaluate_offset_inequality(timespan_start, timespan_stop, offset)`

`(TypedList).extend(tokens)`

Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

`CompoundInequality.get_offset_indices(timespan_1, timespan_2_start_offsets, timespan_2_stop_offsets)`

`(TypedList).index(token)`

Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('cqr', 'as', 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as")
1
```

Returns index.

`(TypedList).insert(i, token)`

Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('1', 2, 4.3))
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

`(TypedList).pop(i=-1)`

Aliases `list.pop()`.

`(TypedList).remove(token)`

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) .**reverse**()

Aliases list.reverse().

(TypedList) .**sort** (*cmp=None, key=None, reverse=False*)

Aliases list.sort().

## Special methods

(TypedCollection) .**\_\_contains\_\_** (*token*)

(TypedList) .**\_\_delitem\_\_** (*i*)

Aliases list.\_\_delitem\_\_().

(TypedCollection) .**\_\_eq\_\_** (*expr*)

(TypedList) .**\_\_getitem\_\_** (*i*)

Aliases list.\_\_getitem\_\_().

(TypedList) .**\_\_iadd\_\_** (*expr*)

Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList (
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList ([
    contexttools.DynamicMark (
        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark (
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark (
        'mp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark (
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark (
        'fff',
        target_context=stafftools.Staff
    )
],
    item_class=contexttools.DynamicMark
)
```

Returns collection.

(TypedCollection) .**\_\_iter\_\_** ()

(TypedCollection) .**\_\_len\_\_** ()

(TypedCollection) .**\_\_ne\_\_** (*expr*)

(AbjadObject) .**\_\_repr\_\_** ()

Interpreter representation of Abjad object.

Returns string.

(TypedList) .**\_\_reversed\_\_** ()

Aliases list.\_\_reversed\_\_().

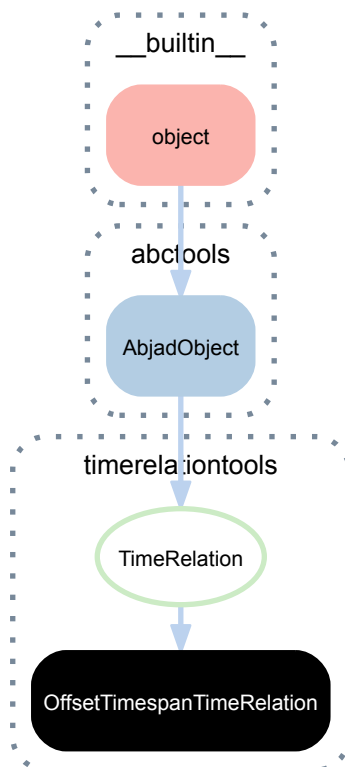
(TypedList) .**\_\_setitem\_\_** (*i, expr*)

Change tokens in *expr* to items and set:

```
>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch('gqs,')
],
  item_class=pitchtools.NamedPitch
)
```

```
>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch("f'"),
  pitchtools.NamedPitch("g'"),
  pitchtools.NamedPitch("a'"),
  pitchtools.NamedPitch("b'"),
  pitchtools.NamedPitch("c'")
],
  item_class=pitchtools.NamedPitch
)
```

### 37.2.2 timerelationtools.OffsetTimespanTimeRelation



**class** timerelationtools.**OffsetTimespanTimeRelation** (*inequality*, *timespan=None*, *offset=None*)

An offset vs. timespan time relation.

```
>>> offset = Offset(5)
>>> timespan = timespantools.Timespan(0, 10)
>>> time_relation = timerelationtools.offset_happens_during_timespan(
...     offset=offset,
...     timespan=timespan,
...     hold=True,
... )
```

```
>>> print time_relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan.start <= offset'),
    timerelationtools.SimpleInequality('offset < timespan.stop')
  ],
  logical_operator='and'
),
  timespan=timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(10, 1)
  ),
  offset=durationtools.Offset(5, 1)
)
```

Offset / timespan time relations are immutable.

## Bases

- `timerelationtools.TimeRelation`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(TimeRelation).**inequality**

Time relation inequality.

OffsetTimespanTimeRelation.**is\_fully\_loaded**

True when *timespan* and *offset* are both not none. Otherwise false:

```
>>> time_relation.is_fully_loaded
True
```

Returns boolean.

OffsetTimespanTimeRelation.**is\_fully\_unloaded**

True when *timespan* and *offset* are both none. Otherwise false:

```
>>> time_relation.is_fully_unloaded
False
```

Returns boolean.

OffsetTimespanTimeRelation.**offset**

Time relation offset:

```
>>> time_relation.offset
Offset(5, 1)
```

Returns offset or none.

OffsetTimespanTimeRelation.**storage\_format**

Time relation storage format:

```
>>> print time_relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan.start <= offset'),
    timerelationtools.SimpleInequality('offset < timespan.stop')
  ],
  logical_operator='and'
),
  timespan=timespantools.Timespan(
```

```

        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    offset=durationtools.Offset(5, 1)
)

```

Returns string.

OffsetTimespanTimeRelation.**timespan**

Time relation timespan:

```

>>> time_relation.timespan
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))

```

Returns timespan or none.

## Methods

(TimeRelation).**new**(\*\*kwargs)

Initialize new time relation with keyword arguments optionally changed:

```

>>> time_relation = \
...     timerelationtools.timespan_2_stops_when_timespan_1_starts()
>>> new_time_relation = \
...     time_relation.new(timespan_1=timespantools.Timespan(0, 5))

>>> print time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
    ],
        logical_operator='and'
    )
)

>>> print new_time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
    ],
        logical_operator='and'
    ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
    )
)

```

Returns newly constructed time relation.

## Special methods

OffsetTimespanTimeRelation.**\_\_call\_\_**(timespan=None, offset=None)

Evaluates time relation:

```

>>> time_relation()
True

```

Raises value error if either *offset* or *timespan* is none.

Otherwise returns boolean.

OffsetTimespanTimeRelation.**\_\_eq\_\_**(expr)

True when *expr* equals time relation. Otherwise false:

```
>>> offset = Offset(5)
>>> time_relation_1 = \
...     timerelationtools.offset_happens_during_timespan()
>>> time_relation_2 = \
...     timerelationtools.offset_happens_during_timespan(
...         offset=offset)
```

```
>>> time_relation_1 == time_relation_1
True
>>> time_relation_1 == time_relation_2
False
>>> time_relation_2 == time_relation_2
True
```

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

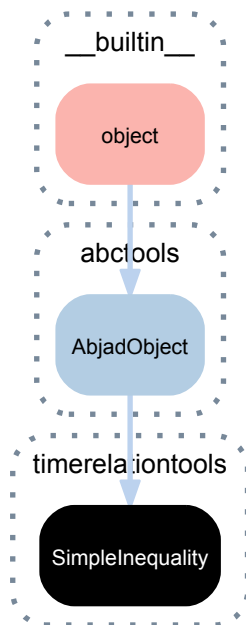
Returns boolean.

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

### 37.2.3 timerelationtools.SimpleInequality



**class** timerelationtools.**SimpleInequality**(*template*)

A simple time-relation inequality.

```
>>> template = 'timespan_2.start_offset < timespan_1.start_offset'
>>> simple_inequality = timerelationtools.SimpleInequality(template)
```

```
>>> simple_inequality
SimpleInequality('timespan_2.start_offset < timespan_1.start_offset')
```

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`SimpleInequality.storage_format`

Simple inequality storage format.

```
>>> print simple_inequality.storage_format
timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset')
```

Returns string.

`SimpleInequality.template`

Simple inequality template.

```
>>> simple_inequality.template
'timespan_2.start_offset < timespan_1.start_offset'
```

Returns string.

## Methods

`SimpleInequality.evaluate` (*timespan\_1\_start\_offset*, *timespan\_1\_stop\_offset*, *timespan\_2\_start\_offset*, *timespan\_2\_stop\_offset*)

`SimpleInequality.evaluate_offset_inequality` (*timespan\_start*, *timespan\_stop*, *offset*)

`SimpleInequality.get_offset_indices` (*timespan\_1*, *timespan\_2\_start\_offsets*, *timespan\_2\_stop\_offsets*)

Change simple inequality to offset indices.

---

### Todo

add example.

---

Returns nonnegative integer pair.

## Special methods

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

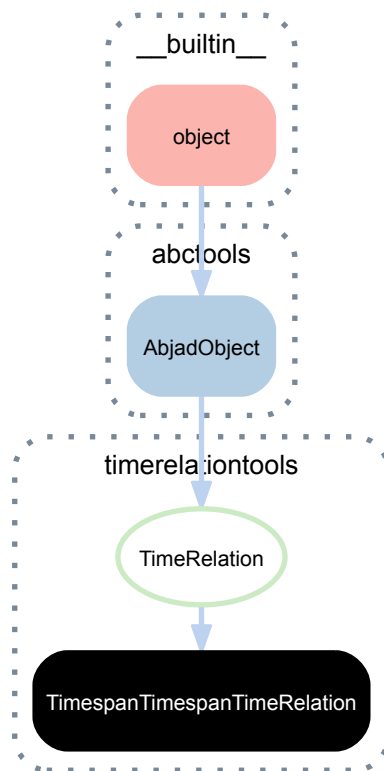
`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.



### 37.2.4 timerelationtools.TimespanTimespanTimeRelation



**class** timerelationtools.**TimespanTimespanTimeRelation** (*inequality*, *timespan\_1=None*,  
*timespan\_2=None*)

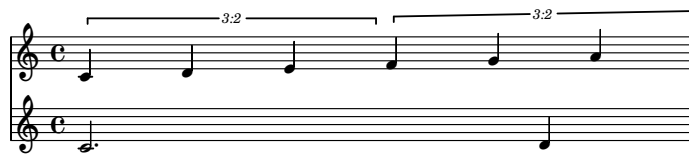
A timespan vs. timespan time relation.

Score for examples:

```
>>> staff_1 = Staff(
...     r"\times 2/3 { c'4 d'4 e'4 } \times 2/3 { f'4 g'4 a'4 }")
>>> staff_2 = Staff("c'2. d'4")
>>> score = Score([staff_1, staff_2])
```

```
>>> last_tuplet = staff_1[-1]
>>> long_note = staff_2[0]
```

```
>>> show(score)
```



**Example 1:**

```
>>> timerelationtools.timespan_2_happens_during_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

**Example 2:**

```
>>> timerelationtools.timespan_2_intersects_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
True
```

**Example 3:**

```
>>> timerelationtools.timespan_2_is_congruent_to_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

#### Example 4:

```
>>> timerelationtools.timespan_2_overlaps_all_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

#### Example 5:

```
>>> timerelationtools.timespan_2_overlaps_start_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
True
```

#### Example 6:

```
>>> timerelationtools.timespan_2_overlaps_stop_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

#### Example 7:

```
>>> timerelationtools.timespan_2_starts_after_timespan_1_starts(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

#### Example 8:

```
>>> timerelationtools.timespan_2_starts_after_timespan_1_stops(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

Timespan / timespan time relations are immutable.

## Bases

- `timerelationtools.TimeRelation`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TimeRelation).inequality`

Time relation inequality.

`TimespanTimespanTimeRelation.is_fully_loaded`

True when *timespan\_1* and *timespan\_2* are both not none. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...         timespan_1=timespan_1, timespan_2=timespan_2, hold=True)
```

```
>>> time_relation.is_fully_loaded
True
```

Returns boolean.

`TimespanTimespanTimeRelation.is_fully_unloaded`

True when *timespan\_1* and *timespan\_2* are both none. Otherwise false.

```
>>> time_relation.is_fully_unloaded
False
```

Returns boolean.

(TimeRelation).**.storage\_format**

Time relation storage format:

```
>>> print time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
  ],
  logical_operator='and'
),
  timespan_1=timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(10, 1)
  ),
  timespan_2=timespantools.Timespan(
    start_offset=durationtools.Offset(5, 1),
    stop_offset=durationtools.Offset(15, 1)
  )
)
```

Returns string.

TimespanTimespanTimeRelation.**.timespan\_1**

Time relation timespan 1:

```
>>> time_relation.timespan_1
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

Returns timespan.

TimespanTimespanTimeRelation.**.timespan\_2**

Time relation timespan 2:

```
>>> time_relation.timespan_2
Timespan(start_offset=Offset(5, 1), stop_offset=Offset(15, 1))
```

Returns timespan.

## Methods

TimespanTimespanTimeRelation.**.get\_counttime\_components** (*counttime\_components*)

Get *counttime\_components* that satisfy *time\_relation*:

```
>>> voice = Voice(
...     [Note(i % 36, Duration(1, 4)) for i in range(200)])
>>> timespan_1 = timespantools.Timespan(20, 22)
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...         timespan_1=timespan_1)
```

```
>>> result = time_relation.get_counttime_components(voice[:])
```

```
>>> for counttime_component in result:
...     counttime_component
Note("af'4")
Note("a'4")
Note("bf'4")
Note("b'4")
Note("c''4")
Note("cs''4")
Note("d'4")
Note("ef''4")
```

```
>>> result.get_timespan()
Timespan(start_offset=Offset(20, 1), stop_offset=Offset(22, 1))
```

*counttime\_components* must belong to a single voice.

*counttime\_components* must be time-contiguous.

The call shown here takes 78355 function calls under r9686.

Returns selection.

TimespanTimespanTimeRelation.**get\_offset\_indices**(*timespan\_2\_start\_offsets*, *timespan\_2\_stop\_offsets*)

Get offset indices that satisfy time relation:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> start_offsets = [inspect(note).get_timespan().start_offset for note in staff]
>>> stop_offsets = [inspect(note).get_timespan().stop_offset for note in staff]
```

**Example 1.** Notes equal to `staff[0:2]` start during timespan `[0, 3/16)`:

```
>>> timespan_1 = timespantools.Timespan(Offset(0), Offset(3, 16))
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...         timespan_1=timespan_1)
>>> time_relation.get_offset_indices(start_offsets, stop_offsets)
(0, 2)
```

**Example 2.** Notes equal to `staff[2:8]` start after timespan `[0, 3/16)` stops:

```
>>> timespan_1 = timespantools.Timespan(Offset(0), Offset(3, 16))
>>> time_relation = \
...     timerelationtools.timespan_2_starts_after_timespan_1_stops(
...         timespan_1=timespan_1)
>>> time_relation.get_offset_indices(start_offsets, stop_offsets)
(2, 8)
```

Returns nonnegative integer pair.

(TimeRelation).**new**(*\*\*kwargs*)

Initialize new time relation with keyword arguments optionally changed:

```
>>> time_relation = \
...     timerelationtools.timespan_2_stops_when_timespan_1_starts()
>>> new_time_relation = \
...     time_relation.new(timespan_1=timespantools.Timespan(0, 5))
```

```
>>> print time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
    ],
    logical_operator='and'
    )
)
```

```
>>> print new_time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
    ],
    logical_operator='and'
    ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
    )
)
```

Returns newly constructed time relation.

## Special methods

`TimespanTimespanTimeRelation.__call__` (*timespan\_1=None, timespan\_2=None*)  
Evaluate time relation.

**Example 1.** Evaluate time relation without substitution:

```
>>> timespan_1 = timespantools.Timespan(5, 15)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1,
...     timespan_2=timespan_2,
...     hold=True,
... )
```

```
>>> print time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
    ],
    logical_operator='and'
),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(15, 1)
    ),
    timespan_2=timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
)
```

```
>>> time_relation()
True
```

**Example 2.** Substitute *timespan\_1* during evaluation:

```
>>> new_timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> new_timespan_1
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

```
>>> time_relation(timespan_1=new_timespan_1)
False
```

**Example 3.** Substitute *timespan\_2* during evaluation:

```
>>> new_timespan_2 = timespantools.Timespan(2, 12)
```

```
>>> new_timespan_2
Timespan(start_offset=Offset(2, 1), stop_offset=Offset(12, 1))
```

```
>>> time_relation(timespan_2=new_timespan_2)
False
```

**Example 4.** Substitute both *timespan\_1* and *timespan\_2* during evaluation:

```
>>> time_relation(
...     timespan_1=new_timespan_1, timespan_2=new_timespan_2)
True
```

Raise value error if either *timespan\_1* or *timespan\_2* is none.

Otherwise return boolean.

`TimespanTimespanTimeRelation.__eq__` (*expr*)  
True when *expr* equals time relation. Otherwise false:

```
>>> timespan = timespantools.Timespan(0, 10)
>>> time_relation_1 = \
...     timerelationtools.timespan_2_starts_during_timespan_1()
>>> time_relation_2 = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan)
```

```
>>> time_relation_1 == time_relation_1
True
>>> time_relation_1 == time_relation_2
False
>>> time_relation_2 == time_relation_2
True
```

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

## 37.3 Functions

### 37.3.1 timerelationtools.offset\_happens\_after\_timespan\_starts

**timerelationtools.offset\_happens\_after\_timespan\_starts** (*timespan=None*, *offset=None*, *hold=False*)

Makes time relation indicating that *offset* happens after *timespan* starts.

```
>>> relation = timerelationtools.offset_happens_after_timespan_starts()
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan.start < offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.2 timerelationtools.offset\_happens\_after\_timespan\_stops

**timerelationtools.offset\_happens\_after\_timespan\_stops** (*timespan=None*, *offset=None*, *hold=False*)

Makes time relation indicating that *offset* happens after *timespan* stops.

```
>>> relation = timerelationtools.offset_happens_after_timespan_stops()
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan.stop < offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.3 timerelationtools.offset\_happens\_before\_timespan\_starts

`timerelationtools.offset_happens_before_timespan_starts` (*timespan=None*,  
*offset=None*,  
*hold=False*)

Makes time relation indicating that *offset* happens before *timespan* starts.

**Example 1.** Makes time relation indicating that *offset* happens before *timespan* starts:

```
>>> relation = timerelationtools.offset_happens_before_timespan_starts()
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('offset < timespan.start')
  ],
    logical_operator='and'
  )
)
```

**Example 2.** Makes time relation indicating that offset 1/2 happens before *timespan* starts:

```
>>> offset = durationtools.Offset(1, 2)
```

```
>>> relation = \
...     timerelationtools.offset_happens_before_timespan_starts(
...     offset=offset)
```

```
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('offset < timespan.start')
  ],
    logical_operator='and'
  ),
  offset=durationtools.Offset(1, 2)
)
```

**Example 3.** Makes time relation indicating that *offset* happens before timespan [2, 8) starts:

```
>>> timespan = timespantools.Timespan(2, 8)
```

```
>>> relation = \
...     timerelationtools.offset_happens_before_timespan_starts(
...     timespan=timespan)
```

```
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('offset < timespan.start')
  ],
    logical_operator='and'
  ),
  timespan=timespantools.Timespan(
    start_offset=durationtools.Offset(2, 1),
    stop_offset=durationtools.Offset(8, 1)
  )
)
```

**Example 4.** Makes time relation indicating that offset 1/2 happens before timespan [2, 8) starts:

```
>>> relation = timerelationtools.offset_happens_before_timespan_starts(
...     timespan=timespan,
...     offset=offset,
...     hold=True,
...     )
```

```
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('offset < timespan.start')
```

```
    ],
    logical_operator='and'
  ),
  timespan=timespantools.Timespan(
    start_offset=durationtools.Offset(2, 1),
    stop_offset=durationtools.Offset(8, 1)
  ),
  offset=durationtools.Offset(1, 2)
)
```

**Example 5.** Evaluates time relation indicating that offset 1/2 happens before timespan [2, 8) starts:

```
>>> timerelationtools.offset_happens_before_timespan_starts(
...     timespan=timespan,
...     offset=offset,
...     hold=False,
... )
True
```

Returns time relation or boolean.

### 37.3.4 timerelationtools.offset\_happens\_before\_timespan\_stops

`timerelationtools.offset_happens_before_timespan_stops` (*timespan=None*, *offset=None*, *hold=False*)

Makes time relation indicating that *offset* happens before *timespan* stops.

```
>>> relation = timerelationtools.offset_happens_before_timespan_stops()
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('offset < timespan.stop')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.5 timerelationtools.offset\_happens\_during\_timespan

`timerelationtools.offset_happens_during_timespan` (*timespan=None*, *offset=None*, *hold=False*)

Makes time relation indicating that *offset* happens during *timespan*.

```
>>> relation = timerelationtools.offset_happens_during_timespan()
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan.start <= offset'),
    timerelationtools.SimpleInequality('offset < timespan.stop')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.6 timerelationtools.offset\_happens\_when\_timespan\_starts

`timerelationtools.offset_happens_when_timespan_starts` (*timespan=None*, *offset=None*, *hold=False*)

Makes time relation indicating that *offset* happens when *timespan* starts.



```
>>> relation = timerelationtools.offset_happens_when_timespan_starts()
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('offset == timespan.start')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.7 timerelationtools.offset\_happens\_when\_timespan\_stops

`timerelationtools.offset_happens_when_timespan_stops` (*timespan=None*, *offset=None*, *hold=False*)

Makes time relation indicating that *offset* happens when *timespan* stops.

```
>>> relation = timerelationtools.offset_happens_when_timespan_stops()
>>> print relation.storage_format
timerelationtools.OffsetTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('offset == timespan.stop')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.8 timerelationtools.timespan\_2\_contains\_timespan\_1\_improperly

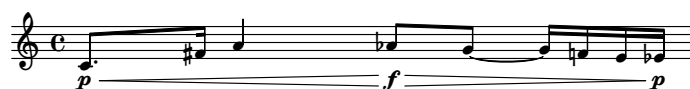
`timerelationtools.timespan_2_contains_timespan_1_improperly` (*timespan\_1=None*, *timespan\_2=None*, *hold=False*)

Makes time relation indicating that *timespan\_2* contains *timespan\_1* improperly.

```
>>> relation = timerelationtools.timespan_2_contains_timespan_1_improperly()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
    timerelationtools.SimpleInequality('timespan_1.stop_offset <= timespan_2.stop_offset')
  ],
  logical_operator='and'
)
```

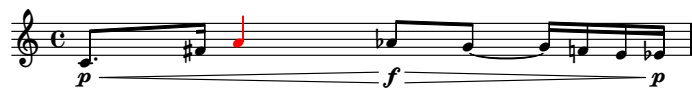
**Example:**

```
>>> staff = Staff(r"c'8. \p \< fs'16 a'4 af'8 \f \> g'8 ~ g'16 f' e' ef' \p")
>>> timespan_1 = timespantools.Timespan(Offset(1, 4), Offset(3, 8))
>>> show(staff)
```



```
>>> offset_lists = staff[:]._get_offset_lists()
>>> time_relation = timerelationtools.timespan_2_contains_timespan_1_improperly(timespan_1=timespan_1)
>>> start_index, stop_index = time_relation.get_offset_indices(*offset_lists)
>>> selected_notes = staff[start_index:stop_index]
>>> selected_notes
SliceSelection(Note("a'4"),)
```

```
>>> labeltools.color_leaves_in_expr(selected_notes, 'red')
>>> show(staff)
```



Returns time relation or boolean.

### 37.3.9 timerelationtools.timespan\_2\_curtails\_timespan\_1

`timerelationtools.timespan_2_curtails_timespan_1` (*timespan\_1=None*, *timespan\_2=None*, *hold=False*)

Makes time relation indicating that *timespan\_2* curtails *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_curtails_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset'),
    timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.stop_offset'),
    timerelationtools.SimpleInequality('timespan_1.stop_offset <= timespan_2.stop_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.10 timerelationtools.timespan\_2\_delays\_timespan\_1

`timerelationtools.timespan_2_delays_timespan_1` (*timespan\_1=None*, *timespan\_2=None*, *hold=False*)

Makes time relation indicating that *timespan\_2* delays *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_delays_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
    timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.11 timerelationtools.timespan\_2\_happens\_during\_timespan\_1

`timerelationtools.timespan_2_happens_during_timespan_1` (*timespan\_1=None*, *timespan\_2=None*, *hold=False*)

Makes time relation indicating that *timespan\_2* happens during *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_happens_during_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
    timerelationtools.SimpleInequality('timespan_2.stop_offset <= timespan_1.stop_offset')
  ],
  logical_operator='and'
)
```

Evaluates whether timespan [7/8, 8/8) happens during timespan [1/2, 3/2):

```
>>> timespan_1 = timespantools.Timespan(Offset(1, 2), Offset(3, 2))
>>> timespan_2 = timespantools.Timespan(Offset(7, 8), Offset(8, 8))
>>> timerelationtools.timespan_2_happens_during_timespan_1(
...     timespan_1=timespan_1,
...     timespan_2=timespan_2,
... )
True
```

Returns time relation or boolean.

### 37.3.12 timerelationtools.timespan\_2\_intersects\_timespan\_1

`timerelationtools.timespan_2_intersects_timespan_1` (*timespan\_1=None, timespan\_2=None, hold=False*)

Makes time relation indicating that *timespan\_2* intersects *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_intersects_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.CompoundInequality([
            timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
            timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
    ),
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
    ],
    logical_operator='and'
    ),
    logical_operator='or'
)
```

Returns time relation or boolean.

### 37.3.13 timerelationtools.timespan\_2\_is\_congruent\_to\_timespan\_1

`timerelationtools.timespan_2_is_congruent_to_timespan_1` (*timespan\_1=None, timespan\_2=None, hold=False*)

Makes time relation indicating that *timespan\_2* is congruent to *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_is_congruent_to_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset == timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.stop_offset == timespan_2.stop_offset')
    ],
    logical_operator='and'
    )
)
```

Returns time relation or boolean.

### 37.3.14 `timerelationtools.timespan_2_overlaps_all_of_timespan_1`

`timerelationtools.timespan_2_overlaps_all_of_timespan_1` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* overlaps all of *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_overlaps_all_of_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
    timerelationtools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.15 `timerelationtools.timespan_2_overlaps_only_start_of_timespan_1`

`timerelationtools.timespan_2_overlaps_only_start_of_timespan_1` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* happens during *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_overlaps_only_start_of_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
    timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
    timerelationtools.SimpleInequality('timespan_2.stop_offset <= timespan_1.stop_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.16 `timerelationtools.timespan_2_overlaps_only_stop_of_timespan_1`

`timerelationtools.timespan_2_overlaps_only_stop_of_timespan_1` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* overlaps only stop of *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_overlaps_only_stop_of_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
    timerelationtools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.17 timerelationtools.timespan\_2\_overlaps\_start\_of\_timespan\_1

`timerelationtools.timespan_2_overlaps_start_of_timespan_1` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* overlaps start of *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_overlaps_start_of_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
    timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
  ],
    logical_operator='and'
  )
)
```

Returns time relation or boolean.

### 37.3.18 timerelationtools.timespan\_2\_overlaps\_stop\_of\_timespan\_1

`timerelationtools.timespan_2_overlaps_stop_of_timespan_1` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Make time relation indicating that *timespan\_2* overlaps stop of *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_overlaps_stop_of_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
    timerelationtools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset')
  ],
    logical_operator='and'
  )
)
```

Returns time relation or boolean.

### 37.3.19 timerelationtools.timespan\_2\_starts\_after\_timespan\_1\_starts

`timerelationtools.timespan_2_starts_after_timespan_1_starts` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* happens during *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_starts_after_timespan_1_starts()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset')
  ],
    logical_operator='and'
  )
)
```

Returns time relation or boolean.

### 37.3.20 `timerelementools.timespan_2_starts_after_timespan_1_stops`

`timerelementools.timespan_2_starts_after_timespan_1_stops` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* starts after *timespan\_1* stops.

```
>>> relation = timerelementools.timespan_2_starts_after_timespan_1_stops()
>>> print relation.storage_format
timerelementools.TimespanTimespanTimeRelation(
  timerelementools.CompoundInequality([
    timerelementools.SimpleInequality('timespan_1.stop_offset <= timespan_2.start_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.21 `timerelementools.timespan_2_starts_before_timespan_1_starts`

`timerelementools.timespan_2_starts_before_timespan_1_starts` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* starts before *timespan\_1* starts.

```
>>> relation = timerelementools.timespan_2_starts_before_timespan_1_starts()
>>> print relation.storage_format
timerelementools.TimespanTimespanTimeRelation(
  timerelementools.CompoundInequality([
    timerelementools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.22 `timerelementools.timespan_2_starts_before_timespan_1_stops`

`timerelementools.timespan_2_starts_before_timespan_1_stops` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* starts before *timespan\_1* stops.

```
>>> relation = timerelementools.timespan_2_starts_before_timespan_1_stops()
>>> print relation.storage_format
timerelementools.TimespanTimespanTimeRelation(
  timerelementools.CompoundInequality([
    timerelementools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.23 timerelationtools.timespan\_2\_starts\_during\_timespan\_1

`timerelationtools.timespan_2_starts_during_timespan_1` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* starts during *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_starts_during_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
    timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
  ],
  logical_operator='and'
)
```

**Example:**

```
>>> staff_1 = Staff("c'4 d'4 e'4 f'4 g'2 c''2")
>>> staff_2 = Staff("c'2 b'2 a'2 g'2")
>>> score = Score([staff_1, staff_2])
>>> show(score)
```



```
>>> start_offsets = [inspect(note).get_timespan().start_offset for note in staff_1]
>>> stop_offsets = [inspect(note).get_timespan().stop_offset for note in staff_1]
```

```
>>> timespan_1 = timespantools.Timespan(Offset(1, 4), Offset(5, 4))
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...         timespan_1=timespan_1)
>>> start_index, stop_index = time_relation.get_offset_indices(
...     start_offsets, stop_offsets)
```

```
>>> selected_notes = staff_1[start_index:stop_index]
>>> selected_notes
SliceSelection(Note("d'4"), Note("e'4"), Note("f'4"), Note("g'2"))
```

```
>>> labeltools.color_leaves_in_expr(selected_notes, 'red')
```

```
>>> show(score)
```



Returns time relation or boolean.

### 37.3.24 timerelationtools.timespan\_2\_starts\_when\_timespan\_1\_starts

`timerelationtools.timespan_2_starts_when_timespan_1_starts` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* starts when *timespan\_1* starts.

```
>>> relation = timerelationtools.timespan_2_starts_when_timespan_1_starts()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
```

```
        timerelationtools.SimpleInequality('timespan_1.start_offset == timespan_2.start_offset')
    ],
    logical_operator='and'
)
)
```

Returns time relation or boolean.

### 37.3.25 timerelationtools.timespan\_2\_starts\_when\_timespan\_1\_stops

`timerelationtools.timespan_2_starts_when_timespan_1_stops` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* happens during *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_starts_when_timespan_1_stops()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.start_offset == timespan_1.stop_offset')
  ],
  logical_operator='and'
)
)
```

Returns time relation or boolean.

### 37.3.26 timerelationtools.timespan\_2\_stops\_after\_timespan\_1\_starts

`timerelationtools.timespan_2_stops_after_timespan_1_starts` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* stops after *timespan\_1* starts.

```
>>> relation = timerelationtools.timespan_2_stops_after_timespan_1_starts()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
  ],
  logical_operator='and'
)
)
```

Returns time relation or boolean.

### 37.3.27 timerelationtools.timespan\_2\_stops\_after\_timespan\_1\_stops

`timerelationtools.timespan_2_stops_after_timespan_1_stops` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* stops after *timespan\_1* stops.

```
>>> relation = timerelationtools.timespan_2_stops_after_timespan_1_stops()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset')
  ],
  logical_operator='and'
)
)
```



Returns time relation or boolean.

### 37.3.28 timerelationtools.timespan\_2\_stops\_before\_timespan\_1\_starts

`timerelationtools.timespan_2_stops_before_timespan_1_starts` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* happens during *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_stops_before_timespan_1_starts()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset < timespan_1.start_offset')
    ],
    logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.29 timerelationtools.timespan\_2\_stops\_before\_timespan\_1\_stops

`timerelationtools.timespan_2_stops_before_timespan_1_stops` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* happens during *timespan\_1*.

```
>>> time_relation = timerelationtools.timespan_2_stops_before_timespan_1_stops()
>>> print time_relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset < timespan_1.stop_offset')
    ],
    logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.30 timerelationtools.timespan\_2\_stops\_during\_timespan\_1

`timerelationtools.timespan_2_stops_during_timespan_1` (*timespan\_1=None*, *timespan\_2=None*, *hold=False*)

Makes time relation indicating that *timespan\_2* stops during *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_stops_during_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
        timerelationtools.SimpleInequality('timespan_2.stop_offset <= timespan_1.stop_offset')
    ],
    logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.31 `timerelationtools.timespan_2_stops_when_timespan_1_starts`

`timerelationtools.timespan_2_stops_when_timespan_1_starts` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* happens during *timespan\_1*.

```
>>> relation = timerelationtools.timespan_2_stops_when_timespan_1_starts()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.32 `timerelationtools.timespan_2_stops_when_timespan_1_stops`

`timerelationtools.timespan_2_stops_when_timespan_1_stops` (*timespan\_1=None*,  
*timespan\_2=None*,  
*hold=False*)

Makes time relation indicating that *timespan\_2* happens during *timespan\_1*.

```
>>> inequality = timerelationtools.timespan_2_stops_when_timespan_1_stops()
>>> print inequality.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.stop_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

### 37.3.33 `timerelationtools.timespan_2_trisects_timespan_1`

`timerelationtools.timespan_2_trisects_timespan_1` (*timespan\_1=None*, *timespan\_2=None*, *hold=False*)

Makes time relation indicating that *timespan\_2* trisects *timespan\_1*.

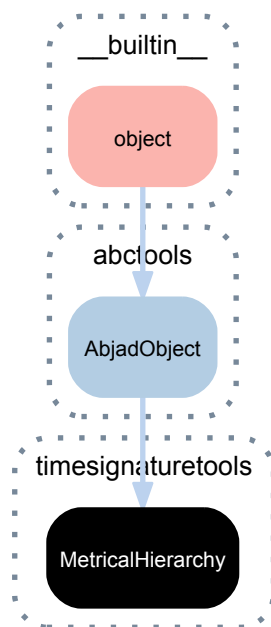
```
>>> relation = timerelationtools.timespan_2_trisects_timespan_1()
>>> print relation.storage_format
timerelationtools.TimespanTimespanTimeRelation(
  timerelationtools.CompoundInequality([
    timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset'),
    timerelationtools.SimpleInequality('timespan_2.stop_offset < timespan_1.stop_offset')
  ],
  logical_operator='and'
)
```

Returns time relation or boolean.

# TIMESIGNATURETOOLS

## 38.1 Concrete classes

### 38.1.1 timesignaturetools.MetricalHierarchy



**class** timesignaturetools.**MetricalHierarchy** (*arg, decrease\_durations\_monotonically=True*)  
A rhythm tree-based model of nested time signature groupings.

The structure of the tree corresponds to the monotonically increasing sequence of factors of the time signature's numerator.

Each deeper level of the tree divides the previous by the next factor in sequence.

Prime divisions greater than 3 are converted to sequences of 2 and 3 summing to that prime. Hence 5 becomes 3+2 and 7 becomes 3+2+2.

The metrical hierarchy models many parts of the common practice understanding of meter:

```
>>> metrical_hierarchy = timesignaturetools.MetricalHierarchy((4, 4))
```

```
>>> metrical_hierarchy
MetricalHierarchy('(4/4 (1/4 1/4 1/4 1/4))')
```

```
>>> print metrical_hierarchy.pretty_rtm_format
(4/4 (
  1/4
  1/4
```



- `__builtin__.object`

## Read-only properties

### `MetricalHierarchy.decrease_durations_monotonically`

True if the metrical hierarchy divides large primes into collections of 2 and 3 that decrease monotonically.

**Example 1.** Metrical hierarchy with durations that increase monotonically:

```
>>> metrical_hierarchy = \
...     timesignaturetools.MetricalHierarchy((5, 4),
...     decrease_durations_monotonically=False)
```

```
>>> metrical_hierarchy.decrease_durations_monotonically
False
```

```
>>> print metrical_hierarchy.pretty_rtm_format
(5/4 (
  (2/4 (
    1/4
    1/4))
  (3/4 (
    1/4
    1/4
    1/4)))
```

**Example 2.** Metrical hierarchy with durations that decrease monotonically:

```
>>> metrical_hierarchy = \
...     timesignaturetools.MetricalHierarchy((5, 4),
...     decrease_durations_monotonically=True)
```

```
>>> metrical_hierarchy.decrease_durations_monotonically
True
```

```
>>> print metrical_hierarchy.pretty_rtm_format
(5/4 (
  (3/4 (
    1/4
    1/4
    1/4))
  (2/4 (
    1/4
    1/4)))
```

Returns boolean.

### `MetricalHierarchy.denominator`

Beat hierarchy denominator:

```
>>> metrical_hierarchy.denominator
4
```

Returns positive integer.

### `MetricalHierarchy.depthwise_offset_inventory`

Depthwise inventory of offsets at each grouping level:

```
>>> for depth, offsets in enumerate(
...     metrical_hierarchy.depthwise_offset_inventory):
...     print depth, offsets
0 (Offset(0, 1), Offset(5, 4))
1 (Offset(0, 1), Offset(3, 4), Offset(5, 4))
2 (Offset(0, 1), Offset(1, 4), Offset(1, 2), Offset(3, 4), Offset(1, 1), Offset(5, 4))
```

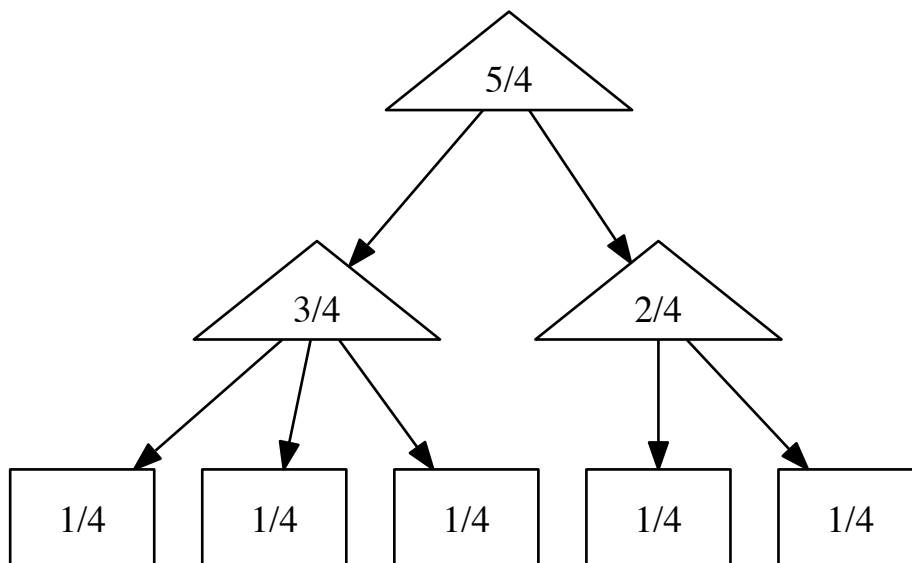
Returns dictionary.

### `MetricalHierarchy.graphviz_format`

Graphviz format of hierarchy's root node:

```
>>> print metrical_hierarchy.graphviz_format
digraph G {
  node_0 [label="5/4",
    shape=triangle];
  node_1 [label="3/4",
    shape=triangle];
  node_2 [label="1/4",
    shape=box];
  node_3 [label="1/4",
    shape=box];
  node_4 [label="1/4",
    shape=box];
  node_5 [label="2/4",
    shape=triangle];
  node_6 [label="1/4",
    shape=box];
  node_7 [label="1/4",
    shape=box];
  node_0 -> node_1;
  node_0 -> node_5;
  node_1 -> node_2;
  node_1 -> node_3;
  node_1 -> node_4;
  node_5 -> node_6;
  node_5 -> node_7;
}
```

```
>>> iotools.graph(metrical_hierarchy)
```



Returns string.

**MetricalHierarchy.implied\_time\_signature**

Implied time signature:

```
>>> timesignaturetools.MetricalHierarchy((4, 4)).implied_time_signature
TimeSignatureMark((4, 4))
```

Returns TimeSignatureMark object.

**MetricalHierarchy.numerator**

Beat hierarchy numerator:

```
>>> metrical_hierarchy.numerator
5
```

Returns positive integer.

**MetricalHierarchy.preprolated\_duration**

Beat hierarchy preprolated\_duration:

```
>>> metrical_hierarchy.preprolated_duration
Duration(5, 4)
```

Returns `preprolated_duration`.

`MetricalHierarchy.pretty_rtm_format`

Beat hierarchy pretty RTM format:

```
>>> print metrical_hierarchy.pretty_rtm_format
(5/4 (
  (3/4 (
    1/4
    1/4
    1/4))
  (2/4 (
    1/4
    1/4))))
```

Returns string.

`MetricalHierarchy.root_node`

Beat hierarchy root node:

```
>>> metrical_hierarchy.root_node
RhythmTreeContainer(
  children=(
    RhythmTreeContainer(
      children=(
        RhythmTreeLeaf(
          preprolated_duration=Duration(1, 4),
          is_pitched=True
        ),
        RhythmTreeLeaf(
          preprolated_duration=Duration(1, 4),
          is_pitched=True
        ),
        RhythmTreeLeaf(
          preprolated_duration=Duration(1, 4),
          is_pitched=True
        )
      ),
      preprolated_duration=NonreducedFraction(3, 4)
    ),
    RhythmTreeContainer(
      children=(
        RhythmTreeLeaf(
          preprolated_duration=Duration(1, 4),
          is_pitched=True
        ),
        RhythmTreeLeaf(
          preprolated_duration=Duration(1, 4),
          is_pitched=True
        )
      ),
      preprolated_duration=NonreducedFraction(2, 4)
    ),
    preprolated_duration=NonreducedFraction(5, 4)
  )
)
```

Returns rhythm tree node.

`MetricalHierarchy.rtm_format`

Beat hierarchy RTM format:

```
>>> metrical_hierarchy.rtm_format
'(5/4 ((3/4 (1/4 1/4 1/4)) (2/4 (1/4 1/4))))'
```

Returns string.

`MetricalHierarchy.storage_format`

Beat hierarchy storage format:

```
>>> print metrical_hierarchy.storage_format
timesignaturetools.MetricalHierarchy(
    '(5/4 ((3/4 (1/4 1/4 1/4)) (2/4 (1/4 1/4))))'
)
```

Returns string.

## Methods

`MetricalHierarchy.generate_offset_kernel_to_denominator(denominator, normalize=True)`

Generate a dictionary of all offsets in a metrical hierarchy up to *denominator*, where the keys are the offsets and the values are the normalized weights of those offsets:

```
>>> metrical_hierarchy = \
...     timesignaturetools.MetricalHierarchy((4, 4))
>>> kernel = \
...     metrical_hierarchy.generate_offset_kernel_to_denominator(8)
>>> for offset, weight in sorted(kernel.kernel.iteritems()):
...     print '{}\t{}'.format(offset, weight)
...
0          3/16
1/8        1/16
1/4        1/8
3/8        1/16
1/2        1/8
5/8        1/16
3/4        1/8
7/8        1/16
1          3/16
```

This is useful for testing how strongly a collection of offsets responds to a given metrical hierarchy.

Returns dictionary.

## Special methods

`MetricalHierarchy.__eq__(expr)`

`MetricalHierarchy.__iter__()`

Iterate metrical hierarchy:

```
>>> metrical_hierarchy = \
...     timesignaturetools.MetricalHierarchy((5, 4))
```

```
>>> for x in metrical_hierarchy:
...     x
...
(NonreducedFraction(0, 4), NonreducedFraction(1, 4))
(NonreducedFraction(1, 4), NonreducedFraction(2, 4))
(NonreducedFraction(2, 4), NonreducedFraction(3, 4))
(NonreducedFraction(0, 4), NonreducedFraction(3, 4))
(NonreducedFraction(3, 4), NonreducedFraction(4, 4))
(NonreducedFraction(4, 4), NonreducedFraction(5, 4))
(NonreducedFraction(3, 4), NonreducedFraction(5, 4))
(NonreducedFraction(0, 4), NonreducedFraction(5, 4))
```

Yield pairs.

`(AbjadObject).__ne__(expr)`

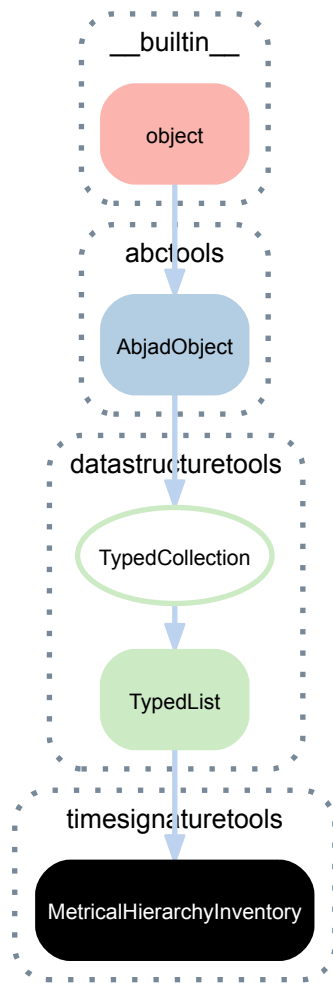
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`MetricalHierarchy.__repr__()`



### 38.1.2 timesignaturetools.MetricalHierarchyInventory



**class** timesignaturetools.**MetricalHierarchyInventory** (*tokens=None*,  
*item\_class=None*,  
*name=None*)

An ordered list of metrical hierarchies.

```
>>> inventory = timesignaturetools.MetricalHierarchyInventory(
...     [(4, 4), (3, 4), (6, 8)])
```

```
>>> print inventory.storage_format
timesignaturetools.MetricalHierarchyInventory([
    timesignaturetools.MetricalHierarchy(
        '(4/4 (1/4 1/4 1/4 1/4))'
    ),
    timesignaturetools.MetricalHierarchy(
        '(3/4 (1/4 1/4 1/4))'
    ),
    timesignaturetools.MetricalHierarchy(
        '(6/8 ((3/8 (1/8 1/8 1/8)) (3/8 (1/8 1/8 1/8))))'
    )
])
```

MetricalHierarchy inventories implement the list interface and are mutable.

#### Bases

- datastructuretools.TypedList
- datastructuretools.TypedCollection

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`(TypedList).append(token)`  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(  
...     item_class=int)  
>>> integer_collection[:]  
[]
```

```
>>> integer_collection.append('1')  
>>> integer_collection.append(2)  
>>> integer_collection.append(3.4)  
>>> integer_collection[:]  
[1, 2, 3]
```

Returns none.

`(TypedList).count(token)`  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(  
...     tokens=[0, 0., '0', 99],  
...     item_class=int)  
>>> integer_collection[:]  
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)  
3
```

Returns count.

`(TypedList).extend(tokens)`  
Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(  
...     item_class=int)  
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))  
>>> integer_collection[:]  
[0, 1, 2, 3]
```

Returns none.

`(TypedList).index(token)`  
Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList (
...     tokens=('cqr', 'as', 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as")
1
```

Returns index.

(TypedList) **.insert** (*i*, *token*)  
Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('1', 2, 4.3))
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) **.new** (*tokens=None*, *item\_class=None*, *name=None*)

(TypedList) **.pop** (*i=-1*)  
Aliases list.pop().

(TypedList) **.remove** (*token*)  
Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

(TypedList) **.reverse** ()  
Aliases list.reverse().

(TypedList) **.sort** (*cmp=None*, *key=None*, *reverse=False*)  
Aliases list.sort().

## Special methods

(TypedCollection) **.\_\_contains\_\_** (*token*)

(TypedList) **.\_\_delitem\_\_** (*i*)  
Aliases list.\_\_delitem\_\_().

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedList) **.\_\_getitem\_\_** (*i*)  
Aliases list.\_\_getitem\_\_().

(TypedList) **.\_\_iadd\_\_** (*expr*)  
Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'fff',
        target_context=stafftools.Staff
    )
],
 item_class=contexttools.DynamicMark
)
```

Returns collection.

(TypedCollection).**\_\_iter\_\_**()

(TypedCollection).**\_\_len\_\_**()

(TypedCollection).**\_\_ne\_\_**(*expr*)

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList).**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList).**\_\_setitem\_\_**(*i*, *expr*)

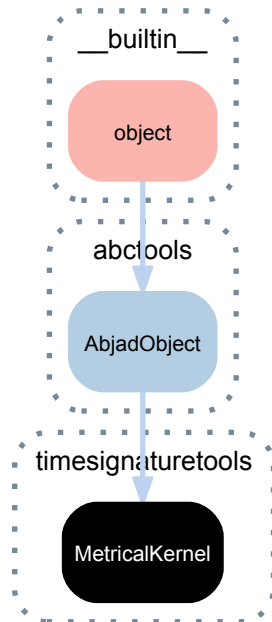
Change tokens in *expr* to items and set:

```
>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c"),
    pitchtools.NamedPitch("d"),
    pitchtools.NamedPitch("e"),
    pitchtools.NamedPitch('gqs,')
],
 item_class=pitchtools.NamedPitch
)
```

```
>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c"),
    pitchtools.NamedPitch("d"),
    pitchtools.NamedPitch("e"),
    pitchtools.NamedPitch("f"),
    pitchtools.NamedPitch("g"),
    pitchtools.NamedPitch("a"),
    pitchtools.NamedPitch("b"),
    pitchtools.NamedPitch("c'")
])
```

```
],
item_class=pitchtools.NamedPitch
)
```

### 38.1.3 timesignaturetools.MetricalKernel



**class** timesignaturetools.**MetricalKernel** (*kernel*)

A metrical kernel, or offset-impulse-response-filter.

```
>>> hierarchy = timesignaturetools.MetricalHierarchy((5, 8))
>>> kernel = hierarchy.generate_offset_kernel_to_denominator(8)
>>> kernel
MetricalKernel({
  Offset(0, 1): Multiplier(3, 11),
  Offset(1, 8): Multiplier(1, 11),
  Offset(1, 4): Multiplier(1, 11),
  Offset(3, 8): Multiplier(2, 11),
  Offset(1, 2): Multiplier(1, 11),
  Offset(5, 8): Multiplier(3, 11)
})
```

Call the kernel against an expression from which offsets can be counted to receive an impulse-response:

```
>>> offsets = [(0, 8), (1, 8), (1, 8), (3, 8)]
>>> kernel(offsets)
0.6363636363636364
```

Return *MetricalKernel* instance.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`MetricalKernel.kernel`

The kernel datastructure.

Returns dict.

## Static methods

`MetricalKernel.count_offsets_in_expr(expr)`

Count offsets in *expr*.

### Example 1:

```
>>> score = Score()
>>> score.append(Staff("c'4. d'8 e'2"))
>>> score.append(Staff(r'\clef bass c4 b,4 a,2'))
```

```
>>> show(score)
```



```
>>> MetricalKernel = timesignaturetools.MetricalKernel
>>> leaves = score.select_leaves(
...     allow_discontiguous_leaves=True)
>>> counter = MetricalKernel.count_offsets_in_expr(leaves)
>>> for offset, count in sorted(counter.items()):
...     offset, count
...
(Offset(0, 1), 2)
(Offset(1, 4), 2)
(Offset(3, 8), 2)
(Offset(1, 2), 4)
(Offset(1, 1), 2)
```

### Example 2:

```
>>> a = timespantools.Timespan(0, 10)
>>> b = timespantools.Timespan(5, 15)
>>> c = timespantools.Timespan(15, 20)
```

```
>>> counter = MetricalKernel.count_offsets_in_expr((a, b, c))
>>> for offset, count in sorted(counter.items()):
...     offset, count
...
(Offset(0, 1), 1)
(Offset(5, 1), 1)
(Offset(10, 1), 1)
(Offset(15, 1), 2)
(Offset(20, 1), 1)
```

Returns counter.

## Special methods

`MetricalKernel.__call__(expr)`

`MetricalKernel.__eq__(expr)`

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`MetricalKernel.__repr__()`

## 38.2 Functions

### 38.2.1 `timesignaturetools.duration_and_possible_denominators_to_time_signature`

`timesignaturetools.duration_and_possible_denominators_to_time_signature` (*duration*,  
*de-*  
*nom-*  
*i-*  
*na-*  
*tors=None*,  
*fac-*  
*tor=None*)

Make new time signature equal to *duration*:

```
>>> timesignaturetools.duration_and_possible_denominators_to_time_signature(  
...     Duration(3, 2))  
TimeSignatureMark((3, 2))
```

Make new time signature equal to *duration* with denominator equal to the first possible element in *denominators*:

```
>>> timesignaturetools.duration_and_possible_denominators_to_time_signature(  
...     Duration(3, 2), denominators=[5, 6, 7, 8])  
TimeSignatureMark((9, 6))
```

Make new time signature equal to *duration* with denominator divisible by *factor*:

```
>>> timesignaturetools.duration_and_possible_denominators_to_time_signature(  
...     Duration(3, 2), factor=5)  
TimeSignatureMark((15, 10))
```

---

**Note:** possibly divide this into two separate functions?

---

Returns new time signature.

### 38.2.2 `timesignaturetools.establish_metrical_hierarchy`

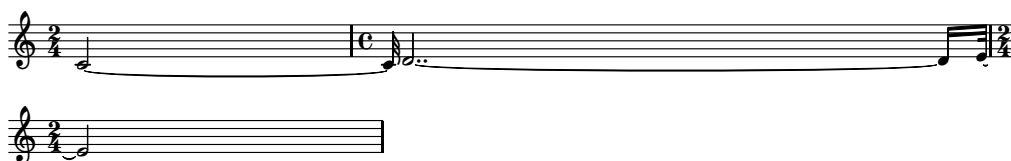
`timesignaturetools.establish_metrical_hierarchy` (*components*, *metrical\_hierarchy*,  
*boundary\_depth=None*, *maximum\_dot\_count=None*)

Rewrite the contents of tie chains in an expression to match a metrical hierarchy.

**Example 1.** Rewrite the contents of a measure in a staff using the default metrical hierarchy for that measure's time signature:

```
>>> parseable = "abj: | 2/4 c'2 ~ |"  
>>> parseable += "| 4/4 c'32 d'2.. ~ d'16 e'32 ~ |"  
>>> parseable += "| 2/4 e'2 |"  
>>> staff = Staff(parseable)
```

```
>>> show(staff)
```

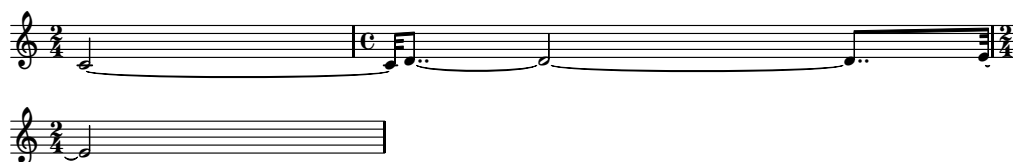


```
>>> hierarchy = timesignaturetools.MetricalHierarchy((4, 4))  
>>> print hierarchy.pretty_rtm_format  
(4/4 (
```

```
1/4
1/4
1/4
1/4))
```

```
>>> timesignaturetools.establish_metrical_hierarchy(
...     staff[1][:],
...     hierarchy,
...     )
```

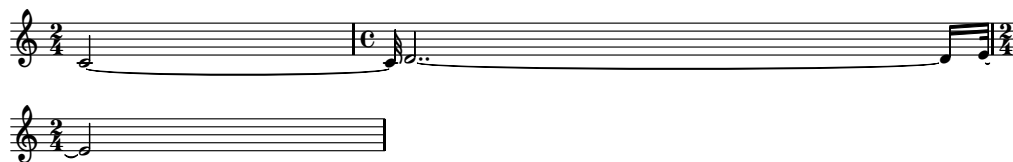
```
>>> show(staff)
```



**Example 2.** Rewrite the contents of a measure in a staff using a custom metrical hierarchy:

```
>>> staff = Staff(parseable)
```

```
>>> show(staff)
```



```
>>> rtm = '(4/4 ((2/4 (1/4 1/4)) (2/4 (1/4 1/4))))'
>>> hierarchy = timesignaturetools.MetricalHierarchy(rtm)
>>> print hierarchy.pretty_rtm_format
(4/4 (
  (2/4 (
    1/4
    1/4))
  (2/4 (
    1/4
    1/4))))
```

```
>>> timesignaturetools.establish_metrical_hierarchy(
...     staff[1][:],
...     hierarchy,
...     )
```

```
>>> show(staff)
```



**Example 3.** Limit the maximum number of dots per leaf using *maximum\_dot\_count*:

```
>>> parseable = "abj: | 3/4 c'32 d'8 e'8 fs'4... |"
>>> measure = p(parseable)
```

```
>>> show(measure)
```



Without constraining the *maximum\_dot\_count*:



```
>>> timesignaturetools.establish_metrical_hierarchy(
...     measure[:],
...     measure,
...     )
```

```
>>> show(measure)
```



Constraining the *maximum\_dot\_count* to 2:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(
...     measure[:],
...     measure,
...     maximum_dot_count=2,
...     )
```

```
>>> show(measure)
```



Constraining the *maximum\_dot\_count* to 1:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(
...     measure[:],
...     measure,
...     maximum_dot_count=1,
...     )
```

```
>>> show(measure)
```



Constraining the *maximum\_dot\_count* to 0:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(
...     measure[:],
...     measure,
...     maximum_dot_count=0,
...     )
```

```
>>> show(measure)
```



**Example 4.** Split tie chains at different depths of the *MetricalHierarchy*, if those tie chains cross any offsets at that depth, but do not also both begin and end at any of those offsets.

Consider the default metrical hierarchy for 9/8:

```
>>> hierarchy = timesignaturetools.MetricalHierarchy((9, 8))
>>> print hierarchy.pretty_rtm_format
(9/8 (
  (3/8 (
    1/8
    1/8
    1/8))
  (3/8 (
    1/8
    1/8
    1/8))
)
```

```
(3/8 (
  1/8
  1/8
  1/8)))
```

We can establish that hierarchy without specifying a *boundary\_depth*:

```
>>> parseable = "abj: | 9/8 c'2 d'2 e'8 |"
>>> measure = p(parseable)
```

```
>>> show(measure)
```



```
>>> timesignaturetools.establish_metrical_hierarchy(
...     measure[:],
...     measure,
... )
```

```
>>> show(measure)
```



With a *boundary\_depth* of 1, tie chains which cross any offsets created by nodes with a depth of 1 in this MetricalHierarchy's rhythm tree - i.e. 0/8, 3/8, 6/8 and 9/8 - which do not also begin and end at any of those offsets, will be split:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(
...     measure[:],
...     measure,
...     boundary_depth=1,
... )
```

```
>>> show(measure)
```



For this 9/8 hierarchy, and this input notation, A *boundary\_depth* of 2 causes no change, as all tie chains already align to multiples of 1/8:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(
...     measure[:],
...     measure,
...     boundary_depth=2,
... )
```

```
>>> show(measure)
```



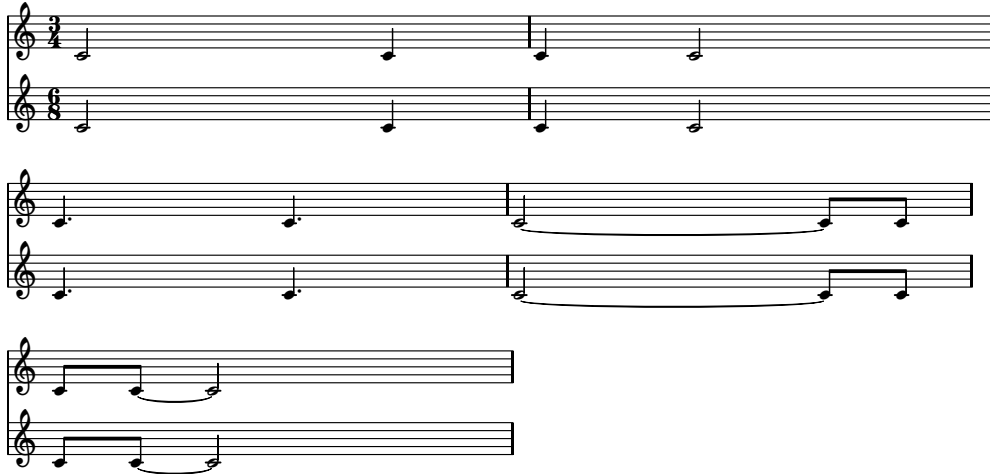
**Example 5.** Comparison of 3/4 and 6/8, at *boundary\_depths* of 0 and 1:

```
>>> triple = "abj: | 3/4 2 4 || 3/4 4 2 || 3/4 4. 4. |"
>>> triple += "| 3/4 2 ~ 8 8 || 3/4 8 8 ~ 2 |"
>>> duples = "abj: | 6/8 2 4 || 6/8 4 2 || 6/8 4. 4. |"
>>> duples += "| 6/8 2 ~ 8 8 || 6/8 8 8 ~ 2 |"
>>> score = Score([Staff(triple), Staff(duples)])
```

In order to see the different time signatures on each staff, we need to move some engravers from the Score context to the Staff context:

```
>>> engravers = ['Timing_translator', 'Time_signature_engraver',
...               'Default_bar_line_engraver']
>>> score.engraver_removals.extend(engravers)
>>> score[0].engraver_consists.extend(engravers)
>>> score[1].engraver_consists.extend(engravers)
```

```
>>> show(score)
```



Here we establish a metrical hierarchy without specifying and boundary depth:

```
>>> for measure in iterationtools.iterate_measures_in_expr(score):
...     timesignaturetools.establish_metrical_hierarchy(
...         measure[:],
...         measure,
...     )
```

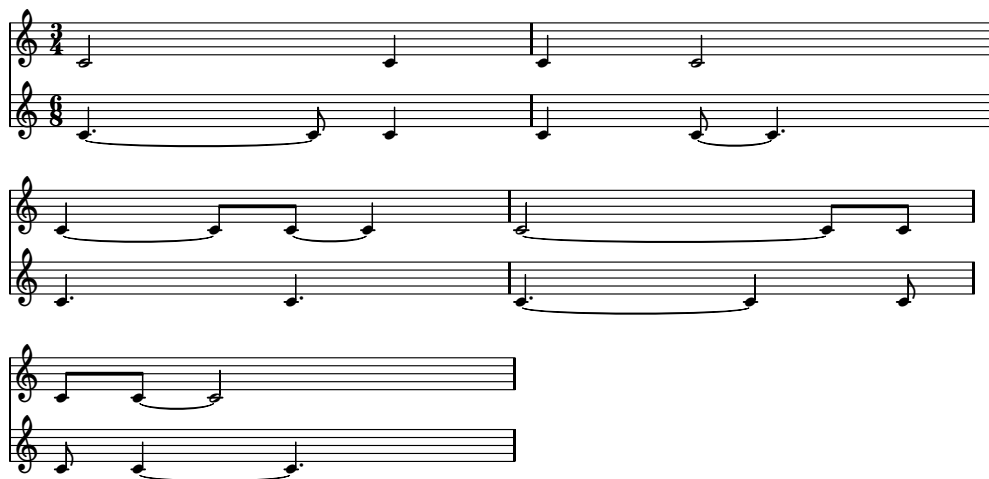
```
>>> show(score)
```



Here we re-establish metrical hierarchy at a boundary depth of 1:

```
>>> for measure in iterationtools.iterate_measures_in_expr(score):
...     timesignaturetools.establish_metrical_hierarchy(
...         measure[:],
...         measure,
...         boundary_depth=1,
...     )
```

```
>>> show(score)
```



Note that the two time signatures are much more clearly disambiguated above.

**Example 6.** Establishing metrical hierarchy recursively in measures with nested tuplets:

```
>>> parseable = "abj: | 4/4 c'16 ~ c'4 d'8. ~ "
>>> parseable += "2/3 { d'8. ~ 3/5 { d'16 e'8. f'16 ~ } } "
>>> parseable += "f'4 |"
>>> measure = p(parseable)
```

```
>>> show(measure)
```



When establishing a metrical hierarchy on a selection of components which contain containers, like *Tuplets* or *Containers*, `timesignaturetools.establish_metrical_hierarchy()` will recurse into those containers, treating them as measures whose time signature is derived from the preprolated `preprolated_duration` of the container's contents:

```
>>> timesignaturetools.establish_metrical_hierarchy(
...     measure[:],
...     measure,
...     boundary_depth=1,
... )
```

```
>>> show(measure)
```



Operates in place and returns none.

### 38.2.3 `timesignaturetools.fit_metrical_hierarchies_to_expr`

```
timesignaturetools.fit_metrical_hierarchies_to_expr(expr, metrical_hierarchies, discard_final_orphan_downbeat=True, starting_offset=None, denominator=32)
```

Find the best-matching sequence of metrical hierarchies for the offsets contained in `expr`.

```
>>> metrical_hierarchies = \
...     timesignaturetools.MetricalHierarchyInventory(
...     [(3, 4), (4, 4), (5, 4)])
```

**Example 1.** Matching a series of hypothetical 4/4 measures:

```
>>> expr = [(0, 4), (4, 4), (8, 4), (12, 4), (16, 4)]
>>> for x in timesignaturetools.fit_metrical_hierarchies_to_expr(
...     expr, metrical_hierarchies):
...     print x.implied_time_signature
...
4/4
4/4
4/4
4/4
```

**Example 2.** Matching a series of hypothetical 5/4 measures:

```
>>> expr = [(0, 4), (3, 4), (5, 4), (10, 4), (15, 4), (20, 4)]
>>> for x in timesignaturetools.fit_metrical_hierarchies_to_expr(
...     expr, metrical_hierarchies):
...     print x.implied_time_signature
...
5/4
5/4
5/4
5/4
```

Offsets are coerced from *expr* via *MetricalKernel.count\_offsets\_in\_expr()*.

MetricalHierarchies are coerced from *metrical\_hierarchies* via *MetricalHierarchyInventory*.

Returns list.

### 38.2.4 timesignaturetools.make\_gridded\_test\_rhythm

`timesignaturetools.make_gridded_test_rhythm(grid_length, rhythm_number, denominator=16)`

Make test rhythm number *rhythm\_number* that fits *grid\_length*.

Returns selection of one or more possibly tied notes.

**Example 1.** The eight test rhythms that fit a length-4 grid:

```
>>> for rhythm_number in range(8):
...     notes = timesignaturetools.make_gridded_test_rhythm(
...         4, rhythm_number, denominator=4)
...     measure = Measure((4, 4), notes)
...     print '{}\t{}'.format(rhythm_number, measure)
...
0 |4/4 c'1|
1 |4/4 c'2. c'4|
2 |4/4 c'2 c'4 c'4|
3 |4/4 c'2 c'2|
4 |4/4 c'4 c'4 c'2|
5 |4/4 c'4 c'4 c'4 c'4|
6 |4/4 c'4 c'2 c'4|
7 |4/4 c'4 c'2. |
```

**Example 2.** The sixteenth test rhythms for that a length-5 grid:

```
>>> for rhythm_number in range(16):
...     notes = timesignaturetools.make_gridded_test_rhythm(
...         5, rhythm_number, denominator=4)
...     measure = Measure((5, 4), notes)
...     print '{}\t{}'.format(rhythm_number, measure)
...
0 |5/4 c'1 ~ c'4|
1 |5/4 c'1 c'4|
2 |5/4 c'2. c'4 c'4|
3 |5/4 c'2. c'2|
4 |5/4 c'2 c'4 c'2|
5 |5/4 c'2 c'4 c'4 c'4|
6 |5/4 c'2 c'2 c'4|
7 |5/4 c'2 c'2. |
```

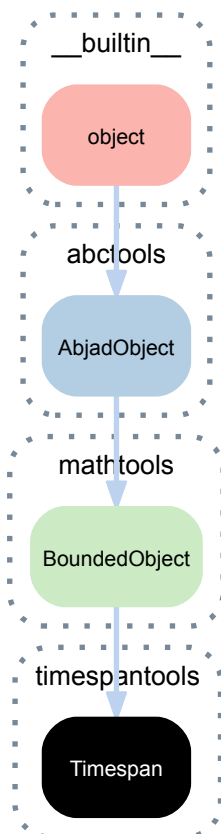
```
8 | 5/4 c'4 c'4 c'2. |  
9 | 5/4 c'4 c'4 c'2 c'4 |  
10 | 5/4 c'4 c'4 c'4 c'4 c'4 |  
11 | 5/4 c'4 c'4 c'4 c'2 |  
12 | 5/4 c'4 c'2 c'2 |  
13 | 5/4 c'4 c'2 c'4 c'4 |  
14 | 5/4 c'4 c'2. c'4 |  
15 | 5/4 c'4 c'1 |
```

Use for testing metrical hierarchy establishment.

# TIMESPANTOOLS

## 39.1 Concrete classes

### 39.1.1 timespantools.Timespan



**class** `timespantools.Timespan` (*start\_offset=NegativeInfinity, stop\_offset=Infinity*)  
A closed-open interval.

**Example:**

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

Timespans are immutable and treated as value objects.

## Bases

- `mathtools.BoundedObject`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `Timespan.axis`

Arithmetic mean of timespan start- and stop-offsets:

```
>>> timespan_1.axis
Offset(5, 1)
```

Returns offset.

### `Timespan.duration`

Get timespan duration:

```
>>> timespan_1.duration
Duration(10, 1)
```

Returns duration.

### `Timespan.is_closed`

False for all timespans:

```
>>> timespan_1.is_closed
False
```

Returns boolean.

### `Timespan.is_half_closed`

True for all timespans:

```
>>> timespan_1.is_half_closed
True
```

Returns boolean.

### `Timespan.is_half_open`

True for all timespans:

```
>>> timespan_1.is_half_open
True
```

Returns boolean.

### `Timespan.is_left_closed`

True for all timespans.

```
>>> timespan_1.is_left_closed
True
```

Returns boolean.

### `Timespan.is_left_open`

False for all timespans.

```
>>> timespan_1.is_left_open
False
```

Returns boolean.

### `Timespan.is_open`

False for all timespans:



```
>>> timespan_1.is_open
False
```

Returns boolean.

**Timespan.is\_right\_closed**  
False for all timespans.

```
>>> timespan_1.is_right_closed
False
```

Returns boolean.

**Timespan.is\_right\_open**  
True for all timespans.

```
>>> timespan_1.is_right_open
True
```

Returns boolean.

**Timespan.is\_well\_formed**  
True when timespan start offset preceeds timespan stop offset. Otherwise false:

```
>>> timespan_1.is_well_formed
True
```

Returns boolean.

**Timespan.offsets**  
Timespan offsets:

```
>>> timespan_1.offsets
(Offset(0, 1), Offset(10, 1))
```

Returns offset pair.

**Timespan.start\_offset**  
Timespan start offset:

```
>>> timespan_1.start_offset
Offset(0, 1)
```

Returns offset.

**Timespan.stop\_offset**  
Timespan stop offset:

```
>>> timespan_1.stop_offset
Offset(10, 1)
```

Returns offset.

**Timespan.storage\_format**  
Timespan storage format:

```
>>> print timespan_1.storage_format
timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(10, 1)
)
```

Returns string.

## Methods

**Timespan.contains\_timespan\_improperly** (*timespan*)  
True when timespan contains *timespan* improperly. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 10)

>>> timespan_1.contains_timespan_improperly(timespan_1)
True
>>> timespan_1.contains_timespan_improperly(timespan_2)
True
>>> timespan_2.contains_timespan_improperly(timespan_1)
False
>>> timespan_2.contains_timespan_improperly(timespan_2)
True
```

Returns boolean.

**Timespan.curtails\_timespan** (*timespan*)

True when timespan curtails *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 10)
```

```
>>> timespan_1.curtails_timespan(timespan_1)
False
>>> timespan_1.curtails_timespan(timespan_2)
False
>>> timespan_2.curtails_timespan(timespan_1)
True
>>> timespan_2.curtails_timespan(timespan_2)
False
```

Returns boolean.

**Timespan.delays\_timespan** (*timespan*)

True when timespan delays *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> timespan_3 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.delays_timespan(timespan_2)
True
>>> timespan_2.delays_timespan(timespan_3)
True
```

Returns boolean.

**Timespan.divide\_by\_ratio** (*ratio*)

Divide timespan by *ratio*:

```
>>> timespan = timespantools.Timespan((1, 2), (3, 2))
```

```
>>> for x in timespan.divide_by_ratio((1, 2, 1)):
...     x
Timespan(start_offset=Offset(1, 2), stop_offset=Offset(3, 4))
Timespan(start_offset=Offset(3, 4), stop_offset=Offset(5, 4))
Timespan(start_offset=Offset(5, 4), stop_offset=Offset(3, 2))
```

Returns tuple of newly constructed timespans.

**Timespan.get\_overlap\_with\_timespan** (*timespan*)

Get duration of overlap with *timespan*:

```
>>> timespan_1 = timespantools.Timespan(0, 15)
>>> timespan_2 = timespantools.Timespan(5, 10)
>>> timespan_3 = timespantools.Timespan(6, 6)
>>> timespan_4 = timespantools.Timespan(12, 22)
```

```
>>> timespan_1.get_overlap_with_timespan(timespan_1)
Duration(15, 1)
```

```
>>> timespan_1.get_overlap_with_timespan(timespan_2)
Duration(5, 1)
```

```
>>> timespan_1.get_overlap_with_timespan(timespan_3)
Duration(0, 1)
```

```
>>> timespan_1.get_overlap_with_timespan(timespan_4)
Duration(3, 1)
```

```
>>> timespan_2.get_overlap_with_timespan(timespan_2)
Duration(5, 1)
```

```
>>> timespan_2.get_overlap_with_timespan(timespan_3)
Duration(0, 1)
```

```
>>> timespan_2.get_overlap_with_timespan(timespan_4)
Duration(0, 1)
```

```
>>> timespan_3.get_overlap_with_timespan(timespan_3)
Duration(0, 1)
```

```
>>> timespan_3.get_overlap_with_timespan(timespan_4)
Duration(0, 1)
```

```
>>> timespan_4.get_overlap_with_timespan(timespan_4)
Duration(10, 1)
```

Returns duration.

**Timespan.happens\_during\_timespan** (*timespan*)

True when timespan happens during *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 10)
```

```
>>> timespan_1.happens_during_timespan(timespan_1)
True
>>> timespan_1.happens_during_timespan(timespan_2)
False
>>> timespan_2.happens_during_timespan(timespan_1)
True
>>> timespan_2.happens_during_timespan(timespan_2)
True
```

Returns boolean.

**Timespan.intersects\_timespan** (*timespan*)

True when timespan intersects *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> timespan_3 = timespantools.Timespan(10, 15)
```

```
>>> timespan_1.intersects_timespan(timespan_1)
True
>>> timespan_1.intersects_timespan(timespan_2)
True
>>> timespan_1.intersects_timespan(timespan_3)
False
```

Returns boolean.

**Timespan.is\_congruent\_to\_timespan** (*timespan*)

True when timespan is congruent to *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.is_congruent_to_timespan(timespan_1)
True
>>> timespan_1.is_congruent_to_timespan(timespan_2)
False
>>> timespan_2.is_congruent_to_timespan(timespan_1)
False
>>> timespan_2.is_congruent_to_timespan(timespan_2)
True
```

Returns boolean.

**Timespan.is\_tangent\_to\_timespan** (*timespan*)  
True when timespan is tangent to *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.is_tangent_to_timespan(timespan_1)
False
>>> timespan_1.is_tangent_to_timespan(timespan_2)
True
>>> timespan_2.is_tangent_to_timespan(timespan_1)
True
>>> timespan_2.is_tangent_to_timespan(timespan_2)
False
```

Returns boolean.

**Timespan.new** (*\*\*kwargs*)  
Create new timespan with *kwargs*:

```
>>> timespan_1.new(stop_offset=Offset(9))
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(9, 1))
```

Returns new timespan.

**Timespan.overlaps\_all\_of\_timespan** (*timespan*)  
True when timespan overlaps all of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 6)
>>> timespan_3 = timespantools.Timespan(5, 10)
```

```
>>> timespan_1.overlaps_all_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_all_of_timespan(timespan_2)
True
>>> timespan_1.overlaps_all_of_timespan(timespan_3)
False
```

Returns boolean.

**Timespan.overlaps\_only\_start\_of\_timespan** (*timespan*)  
True when timespan overlaps only start of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_only_start_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_only_start_of_timespan(timespan_2)
False
>>> timespan_1.overlaps_only_start_of_timespan(timespan_3)
False
>>> timespan_1.overlaps_only_start_of_timespan(timespan_4)
True
```

Returns boolean.

`Timespan.overlaps_only_stop_of_timespan(timespan)`

True when timespan overlaps only stop of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_only_stop_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_only_stop_of_timespan(timespan_2)
True
>>> timespan_1.overlaps_only_stop_of_timespan(timespan_3)
False
>>> timespan_1.overlaps_only_stop_of_timespan(timespan_4)
False
```

Returns boolean.

`Timespan.overlaps_start_of_timespan(timespan)`

True when timespan overlaps start of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_start_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_start_of_timespan(timespan_2)
False
>>> timespan_1.overlaps_start_of_timespan(timespan_3)
True
>>> timespan_1.overlaps_start_of_timespan(timespan_4)
True
```

Returns boolean.

`Timespan.overlaps_stop_of_timespan(timespan)`

True when timespan overlaps stop of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_stop_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_stop_of_timespan(timespan_2)
True
>>> timespan_1.overlaps_stop_of_timespan(timespan_3)
True
>>> timespan_1.overlaps_stop_of_timespan(timespan_4)
False
```

Returns boolean.

`Timespan.reflect(axis=None)`

Reverse timespan about *axis*.

**Example 1.** Reverse timespan about timespan axis:

```
>>> timespantools.Timespan(3, 6).reflect()
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(6, 1))
```

**Example 2.** Reverse timespan about arbitrary axis:

```
>>> timespantools.Timespan(3, 6).reflect(axis=Offset(10))
Timespan(start_offset=Offset(14, 1), stop_offset=Offset(17, 1))
```

Emit newly constructed timespan.

`Timespan.round_offsets` (*multiplier*, *anchor=Left*, *must\_be\_well\_formed=True*)  
Round timespan offsets to multiple of *multiplier*:

```
>>> timespan = timespantools.Timespan((1, 5), (4, 5))
```

```
>>> timespan.round_offsets(1)
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(1, 1))
```

```
>>> timespan.round_offsets(2)
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(2, 1))
```

```
>>> timespan.round_offsets(
...     2,
...     anchor=Right,
... )
Timespan(start_offset=Offset(-2, 1), stop_offset=Offset(0, 1))
```

```
>>> timespan.round_offsets(
...     2,
...     anchor=Right,
...     must_be_well_formed=False,
... )
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(0, 1))
```

Emit newly constructed timespan.

`Timespan.scale` (*multiplier*, *anchor=Left*)  
Scale timespan by *multiplier*.

```
>>> timespan = timespantools.Timespan(3, 6)
```

**Example 1.** Scale timespan relative to timespan start offset:

```
>>> timespan.scale(Multiplier(2))
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(9, 1))
```

**Example 2.** Scale timespan relative to timespan stop offset:

```
>>> timespan.scale(Multiplier(2), anchor=Right)
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(6, 1))
```

Emit newly constructed timespan.

`Timespan.set_duration` (*duration*)  
Set timespan duration to *duration*:

```
>>> timespan = timespantools.Timespan((1, 2), (3, 2))
```

```
>>> timespan.set_duration(Duration(3, 5))
Timespan(start_offset=Offset(1, 2), stop_offset=Offset(11, 10))
```

Emit newly constructed timespan.

`Timespan.set_offsets` (*start\_offset=None*, *stop\_offset=None*)  
Set timespan start offset to *start\_offset* and stop offset to *stop\_offset*:

```
>>> timespan = timespantools.Timespan((1, 2), (3, 2))
```

```
>>> timespan.set_offsets(stop_offset=Offset(7, 8))
Timespan(start_offset=Offset(1, 2), stop_offset=Offset(7, 8))
```

Subtract negative *start\_offset* from existing stop offset:

```
>>> timespan.set_offsets(start_offset=Offset(-1, 2))
Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 2))
```

Subtract negative *stop\_offset* from existing stop offset:

```
>>> timespan.set_offsets(stop_offset=Offset(-1, 2))
Timespan(start_offset=Offset(1, 2), stop_offset=Offset(1, 1))
```

Emit newly constructed timespan.

`Timespan.split_at_offset(offset)`

Split into two parts when *offset* happens during timespan:

```
>>> timespan = timespantools.Timespan(0, 5)
```

```
>>> left, right = timespan.split_at_offset(Offset(2))
```

```
>>> left
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(2, 1))
```

```
>>> right
Timespan(start_offset=Offset(2, 1), stop_offset=Offset(5, 1))
```

Otherwise return a copy of timespan:

```
>>> timespan.split_at_offset(Offset(12))
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(5, 1))
```

Returns one or two newly constructed timespans.

`Timespan.starts_after_offset(offset)`

True when timespan overlaps start of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_after_offset(Offset(-5))
True
>>> timespan_1.starts_after_offset(Offset(0))
False
>>> timespan_1.starts_after_offset(Offset(5))
False
```

Returns boolean.

`Timespan.starts_after_timespan_starts(timespan)`

True when timespan starts after *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_after_timespan_starts(timespan_1)
False
>>> timespan_1.starts_after_timespan_starts(timespan_2)
False
>>> timespan_2.starts_after_timespan_starts(timespan_1)
True
>>> timespan_2.starts_after_timespan_starts(timespan_2)
False
```

Returns boolean.

`Timespan.starts_after_timespan_stops(timespan)`

True when timespan starts after *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> timespan_3 = timespantools.Timespan(10, 20)
>>> timespan_4 = timespantools.Timespan(15, 25)
```

```
>>> timespan_1.starts_after_timespan_stops(timespan_1)
False
>>> timespan_2.starts_after_timespan_stops(timespan_1)
False
>>> timespan_3.starts_after_timespan_stops(timespan_1)
```

```
True
>>> timespan_4.starts_after_timespan_stops(timespan_1)
True
```

Returns boolean.

**Timespan.starts\_at\_offset** (*offset*)

True when timespan starts at *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_at_offset(Offset(-5))
False
>>> timespan_1.starts_at_offset(Offset(0))
True
>>> timespan_1.starts_at_offset(Offset(5))
False
```

Returns boolean.

**Timespan.starts\_at\_or\_after\_offset** (*offset*)

True when timespan starts at or after *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_at_or_after_offset(Offset(-5))
True
>>> timespan_1.starts_at_or_after_offset(Offset(0))
True
>>> timespan_1.starts_at_or_after_offset(Offset(5))
False
```

Returns boolean.

**Timespan.starts\_before\_offset** (*offset*)

True when timespan starts before *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_before_offset(Offset(-5))
False
>>> timespan_1.starts_before_offset(Offset(0))
False
>>> timespan_1.starts_before_offset(Offset(5))
True
```

Returns boolean.

**Timespan.starts\_before\_or\_at\_offset** (*offset*)

True when timespan starts before or at *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_before_or_at_offset(Offset(-5))
False
>>> timespan_1.starts_before_or_at_offset(Offset(0))
True
>>> timespan_1.starts_before_or_at_offset(Offset(5))
True
```

Returns boolean.

**Timespan.starts\_before\_timespan\_starts** (*timespan*)

True when timespan starts before *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```



```
>>> timespan_1.starts_before_timespan_starts(timespan_1)
False
>>> timespan_1.starts_before_timespan_starts(timespan_2)
True
>>> timespan_2.starts_before_timespan_starts(timespan_1)
False
>>> timespan_2.starts_before_timespan_starts(timespan_2)
False
```

Returns boolean.

Timespan.**starts\_before\_timespan\_stops** (*timespan*)

True when timespan starts before *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_before_timespan_stops(timespan_1)
True
>>> timespan_1.starts_before_timespan_stops(timespan_2)
True
>>> timespan_2.starts_before_timespan_stops(timespan_1)
True
>>> timespan_2.starts_before_timespan_stops(timespan_2)
True
```

Returns boolean.

Timespan.**starts\_during\_timespan** (*timespan*)

True when timespan starts during *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_during_timespan(timespan_1)
True
>>> timespan_1.starts_during_timespan(timespan_2)
False
>>> timespan_2.starts_during_timespan(timespan_1)
True
>>> timespan_2.starts_during_timespan(timespan_2)
True
```

Returns boolean.

Timespan.**starts\_when\_timespan\_starts** (*timespan*)

True when timespan starts when *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_when_timespan_starts(timespan_1)
True
>>> timespan_1.starts_when_timespan_starts(timespan_2)
False
>>> timespan_2.starts_when_timespan_starts(timespan_1)
False
>>> timespan_2.starts_when_timespan_starts(timespan_2)
True
```

Returns boolean.

Timespan.**starts\_when\_timespan\_stops** (*timespan*)

True when timespan starts when *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.starts_when_timespan_stops(timespan_1)
False
```

```
>>> timespan_1.starts_when_timespan_stops(timespan_2)
False
>>> timespan_2.starts_when_timespan_stops(timespan_1)
True
>>> timespan_2.starts_when_timespan_stops(timespan_2)
False
```

Returns boolean.

Timespan.**stops\_after\_offset** (*offset*)

True when timespan stops after *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_after_offset(Offset(-5))
True
>>> timespan_1.starts_after_offset(Offset(0))
False
>>> timespan_1.starts_after_offset(Offset(5))
False
```

Returns boolean.

Timespan.**stops\_after\_timespan\_starts** (*timespan*)

True when timespan stops when *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_after_timespan_starts(timespan_1)
True
>>> timespan_1.stops_after_timespan_starts(timespan_2)
False
>>> timespan_2.stops_after_timespan_starts(timespan_1)
True
>>> timespan_2.stops_after_timespan_starts(timespan_2)
True
```

Returns boolean.

Timespan.**stops\_after\_timespan\_stops** (*timespan*)

True when timespan stops when *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_after_timespan_stops(timespan_1)
False
>>> timespan_1.stops_after_timespan_stops(timespan_2)
False
>>> timespan_2.stops_after_timespan_stops(timespan_1)
True
>>> timespan_2.stops_after_timespan_stops(timespan_2)
False
```

Returns boolean.

Timespan.**stops\_at\_offset** (*offset*)

True when timespan stops at *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.stops_at_offset(Offset(-5))
False
>>> timespan_1.stops_at_offset(Offset(0))
False
>>> timespan_1.stops_at_offset(Offset(5))
False
```

Returns boolean.

`Timespan.stops_at_or_after_offset` (*offset*)

True when timespan stops at or after *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)

>>> timespan_1.stops_at_or_after_offset(Offset(-5))
True
>>> timespan_1.stops_at_or_after_offset(Offset(0))
True
>>> timespan_1.stops_at_or_after_offset(Offset(5))
True
```

Returns boolean.

`Timespan.stops_before_offset` (*offset*)

True when timespan stops before *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)

>>> timespan_1.stops_before_offset(Offset(-5))
False
>>> timespan_1.stops_before_offset(Offset(0))
False
>>> timespan_1.stops_before_offset(Offset(5))
False
```

Returns boolean.

`Timespan.stops_before_or_at_offset` (*offset*)

True when timespan stops before or at *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)

>>> timespan_1.stops_before_or_at_offset(Offset(-5))
False
>>> timespan_1.stops_before_or_at_offset(Offset(0))
False
>>> timespan_1.stops_before_or_at_offset(Offset(5))
False
```

Returns boolean.

`Timespan.stops_before_timespan_starts` (*timespan*)

True when timespan stops before *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)

>>> timespan_1.stops_before_timespan_starts(timespan_1)
False
>>> timespan_1.stops_before_timespan_starts(timespan_2)
False
>>> timespan_2.stops_before_timespan_starts(timespan_1)
False
>>> timespan_2.stops_before_timespan_starts(timespan_2)
False
```

Returns boolean.

`Timespan.stops_before_timespan_stops` (*timespan*)

True when timespan stops before *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)

>>> timespan_1.stops_before_timespan_stops(timespan_1)
False
>>> timespan_1.stops_before_timespan_stops(timespan_2)
True
>>> timespan_2.stops_before_timespan_stops(timespan_1)
False
```

```
False
>>> timespan_2.stops_before_timespan_stops(timespan_2)
False
```

Returns boolean.

**Timespan.stops\_during\_timespan** (*timespan*)

True when timespan stops during *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_during_timespan(timespan_1)
True
>>> timespan_1.stops_during_timespan(timespan_2)
False
>>> timespan_2.stops_during_timespan(timespan_1)
False
>>> timespan_2.stops_during_timespan(timespan_2)
True
```

Returns boolean.

**Timespan.stops\_when\_timespan\_starts** (*timespan*)

True when timespan stops when *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_when_timespan_starts(timespan_1)
False
>>> timespan_1.stops_when_timespan_starts(timespan_2)
True
>>> timespan_2.stops_when_timespan_starts(timespan_1)
False
>>> timespan_2.stops_when_timespan_starts(timespan_2)
False
```

Returns boolean.

**Timespan.stops\_when\_timespan\_stops** (*timespan*)

True when timespan stops when *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_when_timespan_stops(timespan_1)
True
>>> timespan_1.stops_when_timespan_stops(timespan_2)
False
>>> timespan_2.stops_when_timespan_stops(timespan_1)
False
>>> timespan_2.stops_when_timespan_stops(timespan_2)
True
```

Returns boolean.

**Timespan.stretch** (*multiplier*, *anchor=None*)

Stretch timespan by *multiplier* relative to *anchor*.

**Example 1.** Stretch relative to timespan start offset:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(2))
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(17, 1))
```

**Example 2.** Stretch relative to timespan stop offset:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(2), Offset(10))
Timespan(start_offset=Offset(-4, 1), stop_offset=Offset(10, 1))
```

**Example 3.** Stretch relative to offset prior to timespan:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(2), Offset(0))
Timespan(start_offset=Offset(6, 1), stop_offset=Offset(20, 1))
```

**Example 4.** Stretch relative to offset after timespan:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(3), Offset(12))
Timespan(start_offset=Offset(-15, 1), stop_offset=Offset(6, 1))
```

**Example 5.** Stretch relative to offset that happens during timespan:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(2), Offset(4))
Timespan(start_offset=Offset(2, 1), stop_offset=Offset(16, 1))
```

Returns newly emitted timespan.

`Timespan.translate(translation=None)`  
Translate timespan by *translation*.

```
>>> timespan = timespantools.Timespan(5, 10)
```

```
>>> timespan.translate(2)
Timespan(start_offset=Offset(7, 1), stop_offset=Offset(12, 1))
```

Emit newly constructed timespan.

`Timespan.translate_offsets(start_offset_translation=None, stop_offset_translation=None)`  
Translate timespan start offset by *start\_offset\_translation* and stop offset by *stop\_offset\_translation*:

```
>>> timespan = timespantools.Timespan((1, 2), (3, 2))
```

```
>>> timespan.translate_offsets(
...     start_offset_translation=Duration(-1, 8))
Timespan(start_offset=Offset(3, 8), stop_offset=Offset(3, 2))
```

Emit newly constructed timespan.

`Timespan.trisects_timespan(timespan)`  
True when timespan trisects *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 6)
```

```
>>> timespan_1.trisects_timespan(timespan_1)
False
>>> timespan_1.trisects_timespan(timespan_2)
False
>>> timespan_2.trisects_timespan(timespan_1)
True
>>> timespan_2.trisects_timespan(timespan_2)
False
```

Returns boolean.

## Special methods

`Timespan.__and__(expr)`  
Logical AND of two timespans:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1 & timespan_2
TimespanInventory([Timespan(start_offset=Offset(5, 1), stop_offset=Offset(10, 1))])
```

```
>>> timespan_1 & timespan_3
TimespanInventory([Timespan(start_offset=Offset(0, 1), stop_offset=Offset(2, 1))])
```

```
>>> timespan_1 & timespan_4
TimespanInventory([])
```

```
>>> timespan_2 & timespan_3
TimespanInventory([])
```

```
>>> timespan_2 & timespan_4
TimespanInventory([Timespan(start_offset=Offset(10, 1), stop_offset=Offset(12, 1))])
```

```
>>> timespan_3 & timespan_4
TimespanInventory([])
```

Returns timespan inventory.

`Timespan.__eq__(timespan)`

True when *timespan* is a timespan with equal offsets:

```
>>> timespantools.Timespan(1, 3) == timespantools.Timespan(1, 3)
True
```

Otherwise false:

```
>>> timespantools.Timespan(1, 3) == timespantools.Timespan(2, 3)
False
```

Returns boolean.

`Timespan.__ge__(expr)`

True when *expr* start offset is greater or equal to timespan start offset:

```
>>> timespan_2 >= timespan_3
True
```

Otherwise false:

```
>>> timespan_1 >= timespan_2
False
```

Returns boolean.

`Timespan.__gt__(expr)`

True when *expr* start offset is greater than timespan start offset:

```
>>> timespan_2 > timespan_3
True
```

Otherwise false:

```
>>> timespan_1 > timespan_2
False
```

Returns boolean.

`Timespan.__le__(expr)`

True when *expr* start offset is less than or equal to timespan start offset:

```
>>> timespan_2 <= timespan_3
False
```

Otherwise false:

```
>>> timespan_1 <= timespan_2
True
```

Returns boolean.

`Timespan.__len__()`

Defined equal to 1 for all timespans:

```
>>> len(timespan_1)
1
```

Returns positive integer.

`Timespan.__lt__(expr)`

True when *expr* start offset is less than timespan start offset:

```
>>> timespan_1 < timespan_2
True
```

Otherwise false:

```
>>> timespan_2 < timespan_3
False
```

Returns boolean.

`Timespan.__ne__(timespan)`

True when *timespan* is not a timespan with equivalent offsets:

```
>>> timespantools.Timespan(1, 3) != timespantools.Timespan(2, 3)
True
```

Otherwise false:

```
>>> timespantools.Timespan(1, 3) != timespantools.Timespan(2/2, (3, 1))
False
```

Returns boolean.

`Timespan.__or__(expr)`

Logical OR of two timespans:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

```
>>> new_timespan = timespan_1 | timespan_2
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

```
>>> new_timespan = timespan_1 | timespan_3
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

```
>>> new_timespan = timespan_1 | timespan_4
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

```
>>> new_timespan = timespan_2 | timespan_3
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

```
>>> new_timespan = timespan_2 | timespan_4
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

```
>>> new_timespan = timespan_3 | timespan_4
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

Returns timespan inventory.

Timespan.\_\_repr\_\_()

Interpreter representation of timespan:

```
>>> timespan_1
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

Returns string.

Timespan.\_\_sub\_\_(*expr*)

Subtract *expr* from timespan:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1 - timespan_1
TimespanInventory([])
```

```
>>> timespan_1 - timespan_2
TimespanInventory([Timespan(start_offset=Offset(0, 1), stop_offset=Offset(5, 1))])
```

```
>>> timespan_1 - timespan_3
TimespanInventory([Timespan(start_offset=Offset(2, 1), stop_offset=Offset(10, 1))])
```

```
>>> timespan_1 - timespan_4
TimespanInventory([Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))])
```

```
>>> timespan_2 - timespan_1
TimespanInventory([Timespan(start_offset=Offset(10, 1), stop_offset=Offset(12, 1))])
```



```
>>> timespan_2 - timespan_2
TimespanInventory([])
```

```
>>> timespan_2 - timespan_3
TimespanInventory([Timespan(start_offset=Offset(5, 1), stop_offset=Offset(12, 1))])
```

```
>>> timespan_2 - timespan_4
TimespanInventory([Timespan(start_offset=Offset(5, 1), stop_offset=Offset(10, 1))])
```

```
>>> timespan_3 - timespan_3
TimespanInventory([])
```

```
>>> timespan_3 - timespan_1
TimespanInventory([Timespan(start_offset=Offset(-2, 1), stop_offset=Offset(0, 1))])
```

```
>>> timespan_3 - timespan_2
TimespanInventory([Timespan(start_offset=Offset(-2, 1), stop_offset=Offset(2, 1))])
```

```
>>> timespan_3 - timespan_4
TimespanInventory([Timespan(start_offset=Offset(-2, 1), stop_offset=Offset(2, 1))])
```

```
>>> timespan_4 - timespan_4
TimespanInventory([])
```

```
>>> timespan_4 - timespan_1
TimespanInventory([Timespan(start_offset=Offset(10, 1), stop_offset=Offset(20, 1))])
```

```
>>> timespan_4 - timespan_2
TimespanInventory([Timespan(start_offset=Offset(12, 1), stop_offset=Offset(20, 1))])
```

```
>>> timespan_4 - timespan_3
TimespanInventory([Timespan(start_offset=Offset(10, 1), stop_offset=Offset(20, 1))])
```

Returns timespan inventory.

Timespan.\_\_xor\_\_(*expr*)

Logical AND of two timespans:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

```
>>> new_timespan = timespan_1 ^ timespan_2
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

```
>>> new_timespan = timespan_1 ^ timespan_3
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(0, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(2, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

```
>>> new_timespan = timespan_1 ^ timespan_4
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

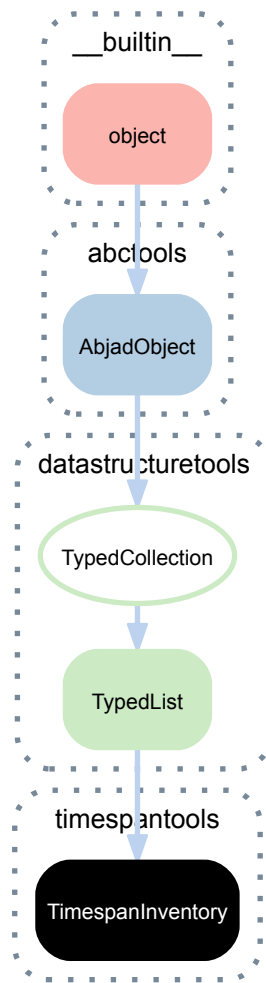
```
>>> new_timespan = timespan_2 ^ timespan_3
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

```
>>> new_timespan = timespan_2 ^ timespan_4
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(12, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

```
>>> new_timespan = timespan_3 ^ timespan_4
>>> print new_timespan.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

Returns timespan inventory.

### 39.1.2 timespantools.TimespanInventory



**class** `timespantools.TimespanInventory` (*tokens=None, item\_class=None, name=None*)  
 A timespan inventory.

#### Example 1:

```
>>> timespan_inventory_1 = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])
```

```
>>> print timespan_inventory_1.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

#### Example 2:

```
>>> timespan_inventory_2 = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(15, 20),
...     ])
```

```
>>> print timespan_inventory_2.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

**Example 3.** Empty timespan inventory:

```
>>> timespan_inventory_3 = timespantools.TimespanInventory()
```

```
>>> print timespan_inventory_3.storage_format
timespantools.TimespanInventory([])
```

Operations on timespan currently work in place.

## Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`TimespanInventory.all_are_contiguous`

True when all timespans are time-contiguous:

```
>>> timespan_inventory_1.all_are_contiguous
True
```

False when timespans not time-contiguous:

```
>>> timespan_inventory_2.all_are_contiguous
False
```

True when empty:

```
>>> timespan_inventory_3.all_are_contiguous
True
```

Returns boolean.

`TimespanInventory.all_are_nonoverlapping`

True when all timespans are non-overlapping:

```
>>> timespan_inventory_1.all_are_nonoverlapping
True
```

False when timespans are overlapping:

```
>>> timespan_inventory_2.all_are_nonoverlapping
False
```

True when empty:

```
>>> timespan_inventory_3.all_are_nonoverlapping
True
```

Returns boolean.

**TimespanInventory.all\_are\_well\_formed**

True when all timespans are well-formed:

```
>>> timespan_inventory_1.all_are_well_formed
True
```

```
>>> timespan_inventory_2.all_are_well_formed
True
```

Also true when empty:

```
>>> timespan_inventory_3.all_are_well_formed
True
```

Otherwise false.

Returns boolean.

**TimespanInventory.axis**

Arithmetic mean of start- and stop-offsets.

```
>>> timespan_inventory_1.axis
Offset(5, 1)
```

```
>>> timespan_inventory_2.axis
Offset(10, 1)
```

None when empty:

```
>>> timespan_inventory_3.axis is None
True
```

Returns offset or none.

**TimespanInventory.duration**

Time from start offset to stop offset:

```
>>> timespan_inventory_1.duration
Duration(10, 1)
```

```
>>> timespan_inventory_2.duration
Duration(20, 1)
```

Zero when empty:

```
>>> timespan_inventory_3.duration
Duration(0, 1)
```

Returns duration.

**TimespanInventory.is\_sorted**

True when timespans are in time order:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
...     ])
```

```
>>> timespan_inventory.is_sorted
True
```

Otherwise false:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(6, 10),
...     timespantools.Timespan(3, 6),
...     ])
```

```
>>> timespan_inventory.is_sorted
False
```

Returns boolean.

(TypedCollection).**item\_class**  
Item class to coerce tokens into.

TimespanInventory.**start\_offset**  
Earliest start offset of any timespan:

```
>>> timespan_inventory_1.start_offset
Offset(0, 1)
```

```
>>> timespan_inventory_2.start_offset
Offset(0, 1)
```

Negative infinity when empty:

```
>>> timespan_inventory_3.start_offset
NegativeInfinity
```

Returns offset or none.

TimespanInventory.**stop\_offset**  
Latest stop offset of any timespan:

```
>>> timespan_inventory_1.stop_offset
Offset(10, 1)
```

```
>>> timespan_inventory_2.stop_offset
Offset(20, 1)
```

Infinity when empty:

```
>>> timespan_inventory_3.stop_offset
Infinity
```

Returns offset or none.

(TypedCollection).**storage\_format**  
Storage format of typed tuple.

TimespanInventory.**timespan**  
Timespan inventory timespan:

```
>>> timespan_inventory_1.timespan
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

```
>>> timespan_inventory_2.timespan
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(20, 1))
```

```
>>> timespan_inventory_3.timespan
Timespan(start_offset=NegativeInfinity, stop_offset=Infinity)
```

Returns timespan.

## Read/write properties

(TypedCollection) .**name**  
Read / write name of typed tuple.

## Methods

(TypedList) .**append** (*token*)  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList (
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

TimespanInventory.**compute\_logical\_and**()  
Compute logical AND of timespans.

### Example 1:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     ])
```

```
>>> result = timespan_inventory.compute_logical_and()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

### Example 2:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
...     ])
```

```
>>> result = timespan_inventory.compute_logical_and()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

### Example 3:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 8),
...     ])
```

```
>>> result = timespan_inventory.compute_logical_and()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(8, 1)
    )
])
```

Same as setwise intersection.

Operates in place and returns timespan inventory.

`TimespanInventory.compute_logical_or()`

Compute logical OR of timespans.

#### Example 1:

```
>>> timespan_inventory = timespantools.TimespanInventory()
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([])
```

#### Example 2:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
... ])
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

#### Example 3:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
... ])
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

#### Example 4:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 2),
... ])
```

```
>>> result = timespan_inventory.compute_logical_or()
```



```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

**Example 5:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(-2, 2),
...     timespantools.Timespan(10, 20),
... ])
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

Operates in place and returns timespan inventory.

`TimespanInventory.compute_logical_xor()`

Compute logical XOR of timespans.

**Example 1:**

```
>>> timespan_inventory = timespantools.TimespanInventory()
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([])
```

**Example 2:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
... ])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

**Example 3:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
... ])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

**Example 4:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 2),
... ])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(0, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(2, 1),
        stop_offset=durationtools.Offset(5, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

**Example 5:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(-2, 2),
...     timespantools.Timespan(10, 20),
... ])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

**Example 6:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(4, 8),
...     timespantools.Timespan(2, 6),
... ])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(2, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(8, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

**Example 7:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(0, 10),
... ])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([])
```

Operates in place and returns timespan inventory.

`TimespanInventory.compute_overlap_factor` (*timespan=None*)  
 Compute overlap factor of timespans:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 15),
...     timespantools.Timespan(20, 25),
...     timespantools.Timespan(20, 30),
... ])
```

**Example 1.** Compute overlap factor across the entire inventory:

```
>>> timespan_inventory.compute_overlap_factor()
Multiplier(7, 6)
```

**Example 2a.** Compute overlap factor within a specific timespan:

```
>>> timespan_inventory.compute_overlap_factor(
...     timespan=timespantools.Timespan(-15, 0))
Multiplier(0, 1)
```

**Example 2b:**

```
>>> timespan_inventory.compute_overlap_factor(
...     timespan=timespantools.Timespan(-10, 5))
Multiplier(1, 3)
```

**Example 2c:**

```
>>> timespan_inventory.compute_overlap_factor(
...     timespan=timespantools.Timespan(-5, 10))
Multiplier(1, 1)
```

**Example 2d:**

```
>>> timespan_inventory.compute_overlap_factor(
...     timespan=timespantools.Timespan(0, 15))
Multiplier(4, 3)
```

**Example 2e:**

```
>>> timespan_inventory.compute_overlap_factor(
...     timespan=timespantools.Timespan(5, 20))
Multiplier(1, 1)
```

**Example 2f:**

```
>>> timespan_inventory.compute_overlap_factor(  
...     timespan=timespantools.Timespan(10, 25))  
Multiplier(1, 1)
```

**Example 2g:**

```
>>> timespan_inventory.compute_overlap_factor(  
...     timespan=timespantools.Timespan(15, 30))  
Multiplier(1, 1)
```

Returns multiplier.

`TimespanInventory.compute_overlap_factor_mapping()`  
Compute overlap factor for each consecutive offset pair in timespans:

```
>>> timespan_inventory = timespantools.TimespanInventory([  
...     timespantools.Timespan(0, 10),  
...     timespantools.Timespan(5, 15),  
...     timespantools.Timespan(20, 25),  
...     timespantools.Timespan(20, 30),  
...     ])  
  
>>> mapping = timespan_inventory.compute_overlap_factor_mapping()  
>>> for timespan, overlap_factor in mapping.iteritems():  
...     timespan.start_offset, timespan.stop_offset, overlap_factor  
...  
(Offset(0, 1), Offset(5, 1), Multiplier(1, 1))  
(Offset(5, 1), Offset(10, 1), Multiplier(2, 1))  
(Offset(10, 1), Offset(15, 1), Multiplier(1, 1))  
(Offset(15, 1), Offset(20, 1), Multiplier(0, 1))  
(Offset(20, 1), Offset(25, 1), Multiplier(2, 1))  
(Offset(25, 1), Offset(30, 1), Multiplier(1, 1))
```

Returns mapping.

`(TypedList).count(token)`  
Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(  
...     tokens=[0, 0., '0', 99],  
...     item_class=int)  
>>> integer_collection[:]  
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)  
3
```

Returns count.

`TimespanInventory.count_offsets()`  
Count offsets in inventory:

**Example 1:**

```
>>> print timespan_inventory_1.storage_format  
timespantools.TimespanInventory([  
    timespantools.Timespan(  
        start_offset=durationtools.Offset(0, 1),  
        stop_offset=durationtools.Offset(3, 1)  
    ),  
    timespantools.Timespan(  
        start_offset=durationtools.Offset(3, 1),  
        stop_offset=durationtools.Offset(6, 1)  
    ),  
    timespantools.Timespan(  
        start_offset=durationtools.Offset(6, 1),  
        stop_offset=durationtools.Offset(10, 1)  
    )  
])
```

```
>>> for offset, count in sorted(
...     timespan_inventory_1.count_offsets().iteritems()):
...     offset, count
...
(Offset(0, 1), 1)
(Offset(3, 1), 2)
(Offset(6, 1), 2)
(Offset(10, 1), 1)
```

**Example 2:**

```
>>> print timespan_inventory_2.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

```
>>> for offset, count in sorted(
...     timespan_inventory_2.count_offsets().iteritems()):
...     offset, count
...
(Offset(0, 1), 1)
(Offset(3, 1), 1)
(Offset(6, 1), 1)
(Offset(10, 1), 1)
(Offset(15, 1), 1)
(Offset(20, 1), 1)
```

**Example 3:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(0, 6),
...     timespantools.Timespan(0, 9),
... ])
>>> for offset, count in sorted(
...     timespan_inventory.count_offsets().iteritems()):
...     offset, count
...
(Offset(0, 1), 3)
(Offset(3, 1), 1)
(Offset(6, 1), 1)
(Offset(9, 1), 1)
```

Returns counter.

`TimespanInventory.explode(inventory_count=None)`

Explode timespans into inventories, avoiding overlap, and distributing density as evenly as possible.

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(5, 13),
...     timespantools.Timespan(6, 10),
...     timespantools.Timespan(8, 9),
...     timespantools.Timespan(15, 23),
...     timespantools.Timespan(16, 21),
...     timespantools.Timespan(17, 19),
...     timespantools.Timespan(19, 20),
...     timespantools.Timespan(25, 30),
...     timespantools.Timespan(26, 29),
...     timespantools.Timespan(32, 34),
... ])
```

```
...    timespantools.Timespan(34, 37),
...    )
```

**Example 1.** Explode timespans into the optimal number of non-overlapping inventories:

```
>>> for exploded_inventory in timespan_inventory.explode():
...     print exploded_inventory.storage_format
...
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(13, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(17, 1),
        stop_offset=durationtools.Offset(19, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(19, 1),
        stop_offset=durationtools.Offset(20, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(34, 1),
        stop_offset=durationtools.Offset(37, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(16, 1),
        stop_offset=durationtools.Offset(21, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(25, 1),
        stop_offset=durationtools.Offset(30, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(8, 1),
        stop_offset=durationtools.Offset(9, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(23, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(26, 1),
        stop_offset=durationtools.Offset(29, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(32, 1),
        stop_offset=durationtools.Offset(34, 1)
    )
])
```

**Example 2.** Explode timespans into a less-than-optimal number of overlapping inventories:

```
>>> for exploded_inventory in timespan_inventory.explode(
...     inventory_count=2):
...     print exploded_inventory.storage_format
...
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
```

```

        stop_offset=durationtools.Offset(13, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(23, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(25, 1),
        stop_offset=durationtools.Offset(30, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(34, 1),
        stop_offset=durationtools.Offset(37, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(8, 1),
        stop_offset=durationtools.Offset(9, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(16, 1),
        stop_offset=durationtools.Offset(21, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(17, 1),
        stop_offset=durationtools.Offset(19, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(19, 1),
        stop_offset=durationtools.Offset(20, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(26, 1),
        stop_offset=durationtools.Offset(29, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(32, 1),
        stop_offset=durationtools.Offset(34, 1)
    )
])

```

**Example 3.** Explode timespans into a greater-than-optimal number of non-overlapping inventories:

```

>>> for exploded_inventory in timespan_inventory.explode(
...     inventory_count=6):
...     print exploded_inventory.storage_format
...
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(16, 1),
        stop_offset=durationtools.Offset(21, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(34, 1),
        stop_offset=durationtools.Offset(37, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(23, 1)
    )
])

```

```
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(8, 1),
        stop_offset=durationtools.Offset(9, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(17, 1),
        stop_offset=durationtools.Offset(19, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(19, 1),
        stop_offset=durationtools.Offset(20, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(26, 1),
        stop_offset=durationtools.Offset(29, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(32, 1),
        stop_offset=durationtools.Offset(34, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(13, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(25, 1),
        stop_offset=durationtools.Offset(30, 1)
    )
])
```

Returns inventories.

(`TypedList`) **.extend**(*tokens*)

Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

`TimespanInventory.get_timespan_that_satisfies_time_relation`(*time\_relation*)

Get timespan that satisfies *time\_relation*:

```
>>> timespan_1 = timespantools.Timespan(2, 5)
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)
```

```
>>> timespan_inventory_1.get_timespan_that_satisfies_time_relation(
...     time_relation)
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(6, 1))
```

Returns timespan when timespan inventory contains exactly one timespan that satisfies *time\_relation*.



Raises exception when timespan inventory contains no timespan that satisfies *time\_relation*.

Raises exception when timespan inventory contains more than one timespan that satisfies *time\_relation*.

`TimespanInventory.get_timespans_that_satisfy_time_relation` (*time\_relation*)

Get timespans that satisfy *time\_relation*:

```
>>> timespan_1 = timespantools.Timespan(2, 8)
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)

>>> result = \
...     timespan_inventory_1.get_timespans_that_satisfy_time_relation(
...     time_relation)

>>> print result.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

Returns new timespan inventory.

`TimespanInventory.has_timespan_that_satisfies_time_relation` (*time\_relation*)

True when timespan inventory has timespan that satisfies *time\_relation*:

```
>>> timespan_1 = timespantools.Timespan(2, 8)
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)

>>> timespan_inventory_1.has_timespan_that_satisfies_time_relation(
...     time_relation)
True
```

Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(10, 20)
>>> time_relation = \
...     timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)

>>> timespan_inventory_1.has_timespan_that_satisfies_time_relation(
...     time_relation)
False
```

Returns boolean.

(`TypedList`) `.index` (*token*)

Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('c'qf', "as'", 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

(`TypedList`) `.insert` (*i*, *token*)

Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['1', 2, 4.3])
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection) .new (tokens=None, item\_class=None, name=None)

TimespanInventory.partition (include\_tangent\_timespans=False)

Partition timespans into inventories:

### Example 1:

```
>>> print timespan_inventory_1.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

```
>>> for inventory in timespan_inventory_1.partition():
...     print inventory.storage_format
...
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

### Example 2:

```
>>> print timespan_inventory_2.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
```

```

        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])

```

```

>>> for inventory in timespan_inventory_2.partition():
...     print inventory.storage_format
...
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])

```

**Example 3.** Treat tangent timespans as part of the same group with `include_tangent_timespans`:

```

>>> for inventory in timespan_inventory_1.partition(
...     include_tangent_timespans=True):
...     print inventory.storage_format
...
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])

```

Returns 0 or more inventories.

(TypedList) **.pop** (*i=-1*)

Aliases `list.pop()`.

`TimespanInventory`. **reflect** (*axis=None*)

Reflect timespans.

**Example 1.** Reflect timespans about timespan inventory axis:

```

>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])

```

```

>>> result = timespan_inventory.reflect()

```

```

>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),

```

```
        stop_offset=durationtools.Offset(4, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(4, 1),
        stop_offset=durationtools.Offset(7, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(7, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

**Example 2. Reflect timespans about arbitrary axis:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])
```

```
>>> result = timespan_inventory.reflect(axis=Offset(15))
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(20, 1),
        stop_offset=durationtools.Offset(24, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(24, 1),
        stop_offset=durationtools.Offset(27, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(27, 1),
        stop_offset=durationtools.Offset(30, 1)
    )
])
```

Operates in place and returns timespan inventory.

(TypedList) . **remove** (*token*)

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(('0', 1.0, 2, 3.14159))
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

TimespanInventory. **remove\_degenerate\_timespans** ()

Remove degenerate timespans:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(5, 5),
...     timespantools.Timespan(5, 10),
...     timespantools.Timespan(5, 25),
... ])
```

```
>>> result = timespan_inventory.remove_degenerate_timespans()
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
```

```

        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(25, 1)
    )
])

```

Operates in place and returns timespan inventory.

`TimespanInventory.repeat_to_stop_offset(stop_offset)`  
 Repeat timespans to *stop\_offset*:

```

>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])

```

```

>>> result = timespan_inventory.repeat_to_stop_offset(15)

```

```

>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(13, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(13, 1),
        stop_offset=durationtools.Offset(15, 1)
    )
])

```

Operates in place and returns timespan inventory.

`(TypedList).reverse()`  
 Aliases `list.reverse()`.

`TimespanInventory.rotate(count)`  
 Rotate by *count* contiguous timespans.

**Example 1.** Rotate by one timespan to the left:

```

>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 4),
...     timespantools.Timespan(4, 10),
... ])

```

```

>>> result = timespan_inventory.rotate(-1)

```

```

>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(1, 1),

```

```
        stop_offset=durationtools.Offset(7, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(7, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

**Example 2.** Rotate by one timespan to the right:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 4),
...     timespantools.Timespan(4, 10),
... ])
```

```
>>> result = timespan_inventory.rotate(1)
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(9, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(9, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

Operates in place and returns timespan inventory.

`TimespanInventory.round_offsets` (*multiplier*, *anchor=Left*, *must\_be\_well\_formed=True*)  
Round offsets of timespans in inventory to multiples of *multiplier*:

**Example 1:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 2),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])
```

```
>>> rounded_inventory = timespan_inventory.round_offsets(3)
>>> print rounded_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(9, 1)
    )
])
```

**Example 2:**

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 2),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])
```

```

>>> rounded_inventory = timespan_inventory.round_offsets(5)
>>> print rounded_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])

```

**Example 3:**

```

>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 2),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])

```

```

>>> rounded_inventory = timespan_inventory.round_offsets(
...     5,
...     anchor=Right,
... )
>>> print rounded_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-5, 1),
        stop_offset=durationtools.Offset(0, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])

```

**Example 4:**

```

>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 2),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])

```

```

>>> rounded_inventory = timespan_inventory.round_offsets(
...     5,
...     anchor=Right,
...     must_be_well_formed=False,
... )
>>> print rounded_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(0, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(5, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])

```

```
)
```

Operates in place and returns timespan inventory.

`TimespanInventory`.**scale** (*multiplier*, *anchor=Left*)

Scale timespan by *multiplier* relative to *anchor*.

**Example 1.** Scale timespans relative to timespan inventory start offset:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
...     ])
```

```
>>> result = timespan_inventory.scale(2)
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(9, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(14, 1)
    )
])
```

**Example 2.** Scale timespans relative to timespan inventory stop offset:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
...     ])
```

```
>>> result = timespan_inventory.scale(2, anchor=Right)
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-3, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(2, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

Operates in place and returns timespan inventory.

(`TypedList`) **.sort** (*cmp=None*, *key=None*, *reverse=False*)

Aliases `list.sort()`.

`TimespanInventory`.**split\_at\_offset** (*offset*)

Split timespans at *offset*:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     ])
```



```
...    timespantools.Timespan(6, 10),
...    ])
```

**Example 1:**

```
>>> left, right = timespan_inventory.split_at_offset(4)
```

```
>>> print left.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(4, 1)
    )
])
```

```
>>> print right.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(4, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

**Example 2:**

```
>>> left, right = timespan_inventory.split_at_offset(6)
```

```
>>> print left.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    )
])
```

```
>>> print right.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

**Example 3:**

```
>>> left, right = timespan_inventory.split_at_offset(-1)
```

```
>>> print left.storage_format
timespantools.TimespanInventory([])
```

```
>>> print right.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
```

```
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

Returns inventories.

`TimespanInventory.split_at_offsets` (*offsets*)

Split timespans at *offsets*:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(4, 10),
...     timespantools.Timespan(15, 20),
...     ])
```

```
>>> offsets = [-1, 3, 6, 12, 13]
```

**Example 1:**

```
>>> for inventory in timespan_inventory.split_at_offsets(offsets):
...     print inventory.storage_format
...
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(4, 1),
        stop_offset=durationtools.Offset(6, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

**Example 2:**

```
>>> timespan_inventory = timespantools.TimespanInventory([])
>>> timespan_inventory.split_at_offsets(offsets)
[TimespanInventory([])]
```

Returns 1 or more inventories.

`TimespanInventory.stretch` (*multiplier*, *anchor=None*)

Stretch timespans by *multiplier* relative to *anchor*.

**Example 1:** Stretch timespans relative to timespan inventory start offset:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
```

```
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
...     ])
```

```
>>> result = timespan_inventory.stretch(2)
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(6, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(12, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(12, 1),
        stop_offset=durationtools.Offset(20, 1)
    )
])
```

**Example 2:** Stretch timespans relative to arbitrary anchor:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
...     ])
```

```
>>> result = timespan_inventory.stretch(2, anchor=Offset(8))
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-8, 1),
        stop_offset=durationtools.Offset(-2, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(4, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(4, 1),
        stop_offset=durationtools.Offset(12, 1)
    )
])
```

Operates in place and returns timespan inventory.

`TimespanInventory.translate` (*translation=None*)

Translate timespans by *translation*.

**Example 1.** Translate timespan by offset 50:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
...     ])
```

```
>>> result = timespan_inventory.translate(50)
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(50, 1),
        stop_offset=durationtools.Offset(53, 1)
    ),
    timespantools.Timespan(
```

```
        start_offset=durationtools.Offset(53, 1),
        stop_offset=durationtools.Offset(56, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(56, 1),
        stop_offset=durationtools.Offset(60, 1)
    )
])
```

Operates in place and returns timespan inventory.

`TimespanInventory.translate_offsets` (*start\_offset\_translation=None*,  
*stop\_offset\_translation=None*)  
Translate timespans by *start\_offset\_translation* and *stop\_offset\_translation*.

**Example 1.** Translate timespan start- and stop-offsets equally:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])
```

```
>>> result = timespan_inventory.translate_offsets(50, 50)
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(50, 1),
        stop_offset=durationtools.Offset(53, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(53, 1),
        stop_offset=durationtools.Offset(56, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(56, 1),
        stop_offset=durationtools.Offset(60, 1)
    )
])
```

**Example 2.** Translate timespan stop-offsets only:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])
```

```
>>> result = timespan_inventory.translate_offsets(
...     stop_offset_translation=20)
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(23, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(26, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(30, 1)
    )
])
```

Operates in place and returns timespan inventory.

## Special methods

`TimespanInventory.__and__(timespan)`

Keep material that intersects *timespan*:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 16),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 8),
...     ])
```

```
>>> timespan = timespantools.Timespan(5, 10)
>>> result = timespan_inventory & timespan
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(8, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
    )
])
```

Operates in place and returns timespan inventory.

`(TypedCollection).__contains__(token)`

`(TypedList).__delitem__(i)`  
 Aliases `list.__delitem__()`.

`(TypedCollection).__eq__(expr)`

`(TypedList).__getitem__(i)`  
 Aliases `list.__getitem__()`.

`(TypedList).__iadd__(expr)`  
 Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'fff',
        target_context=stafftools.Staff
    )
])
```

```
)
],
item_class=contexttools.DynamicMark
)
```

Returns collection.

(TypedCollection) .**\_\_iter\_\_**()

(TypedCollection) .**\_\_len\_\_**()

(TypedCollection) .**\_\_ne\_\_**(*expr*)

(AbjadObject) .**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

(TypedList) .**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

(TypedList) .**\_\_setitem\_\_**(*i*, *expr*)

Change tokens in *expr* to items and set:

```
>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch('gqs,')
],
item_class=pitchtools.NamedPitch
)
```

```
>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch("f'"),
  pitchtools.NamedPitch("g'"),
  pitchtools.NamedPitch("a'"),
  pitchtools.NamedPitch("b'"),
  pitchtools.NamedPitch("c'")
],
item_class=pitchtools.NamedPitch
)
```

TimespanInventory .**\_\_sub\_\_**(*timespan*)

Delete material that intersects *timespan*:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 16),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 8),
... ])
```

```
>>> timespan = timespantools.Timespan(5, 10)
>>> result = timespan_inventory - timespan
```

```
>>> print timespan_inventory.storage_format
timespantools.TimespanInventory([
  timespantools.Timespan(
    start_offset=durationtools.Offset(-2, 1),
    stop_offset=durationtools.Offset(5, 1)
  ),
  timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(5, 1)
  )
])
```

```
    ),  
    timespantools.Timespan(  
        start_offset=durationtools.Offset(10, 1),  
        stop_offset=durationtools.Offset(12, 1)  
    ),  
    timespantools.Timespan(  
        start_offset=durationtools.Offset(10, 1),  
        stop_offset=durationtools.Offset(16, 1)  
    )  
])
```

Operates in place and returns timespan inventory.

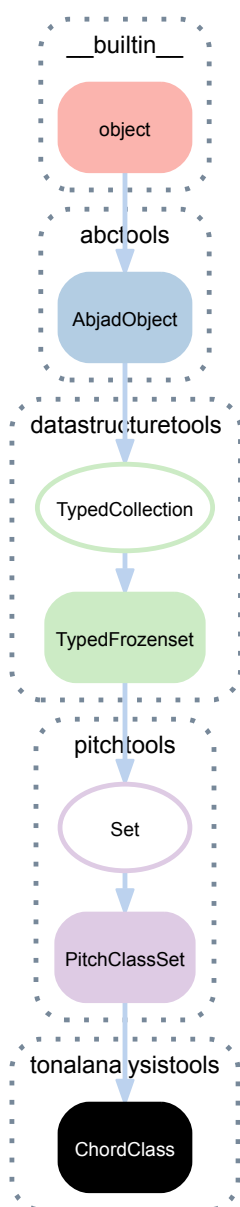




# TONALANALYSISTOOLS

## 40.1 Concrete classes

### 40.1.1 tonalanalysistools.ChordClass



**class** `tonalanalysistools.ChordClass` (*root*, \**args*)

A tonal chord class such as G 7, G 6/5, G half-diminished 6/5, etc.

Note that notions like G 7 represent an entire *class of* chords because there are many different spacings and registrations of a G 7 chord.

## Bases

- `pitchtools.PitchClassSet`
- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ChordClass.bass`

`ChordClass.cardinality`

`ChordClass.extent`

`ChordClass.figured_bass`

`ChordClass.inversion`

`(TypedCollection).item_class`

Item class to coerce tokens into.

`ChordClass.markup`

`ChordClass.quality_indicator`

`ChordClass.quality_pair`

`ChordClass.root`

`ChordClass.root_string`

`(TypedCollection).storage_format`

Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`

Read / write name of typed tuple.

## Methods

`(TypedFrozenSet).copy()`

`(TypedFrozenSet).difference(expr)`

`(TypedFrozenSet).intersection(expr)`

`(PitchClassSet).invert()`

Invert numbered pitch-class set:

```
>>> pitchtools.PitchClassSet (
...     [-2, -1.5, 6, 7, -1.5, 7],
...     ).invert ()
PitchClassSet ([1.5, 2, 5, 6])
```

Returns numbered pitch-class set.

(PitchClassSet) **.is\_transposed\_subset** (*pcset*)  
 True when self is transposed subset of *pcset*. False otherwise:

```
>>> pitch_class_set_1 = pitchtools.PitchClassSet (
...     [-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_set_2 = pitchtools.PitchClassSet (
...     [-2, -1.5, 6, 7, -1.5, 7, 7.5, 8],
...     )
```

```
>>> pitch_class_set_1.is_transposed_subset (pitch_class_set_2)
True
```

Returns boolean.

(PitchClassSet) **.is\_transposed\_superset** (*pcset*)  
 True when self is transposed superset of *pcset*. False otherwise:

```
>>> pitch_class_set_1 = pitchtools.PitchClassSet (
...     [-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_set_2 = pitchtools.PitchClassSet (
...     [-2, -1.5, 6, 7, -1.5, 7, 7.5, 8],
...     )
```

```
>>> pitch_class_set_2.is_transposed_superset (pitch_class_set_1)
True
```

Returns boolean.

(TypedFrozenset) **.isdisjoint** (*expr*)

(TypedFrozenset) **.issubset** (*expr*)

(TypedFrozenset) **.issuperset** (*expr*)

(PitchClassSet) **.multiply** (*n*)  
 Multiply pitch-class set by *n*:

```
>>> pitchtools.PitchClassSet (
...     [-2, -1.5, 6, 7, -1.5, 7],
...     ).multiply (5)
PitchClassSet ([2, 4.5, 6, 11])
```

Returns numbered pitch-class set.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

(PitchClassSet) **.order\_by** (*pitch\_class\_segment*)

(TypedFrozenset) **.symmetric\_difference** (*expr*)

ChordClass **.transpose** ()

(TypedFrozenset) **.union** (*expr*)

## Class methods

(PitchClassSet) **.from\_selection** (*selection, item\_class=None, name=None*)  
 Initialize pitch-class set from component selection:

```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> pitchtools.PitchClassSet.from_selection(selection)
PitchClassSet(['c', 'd', 'fs', 'g', 'a', 'b'])
```

Returns pitch-class set.

## Static methods

`ChordClass.cardinality_to_extent` (*cardinality*)

Change *cardinality* to extent.

**Example:** tertian chord with four pitch classes qualifies as a seventh chord:

```
>>> tonalanalysistools.ChordClass.cardinality_to_extent(4)
7
```

Returns integer.

`ChordClass.extent_to_cardinality` (*extent*)

Change *extent* to cardinality.

**Example:** tertian chord with extent of seven comprises four pitch-classes:

```
>>> tonalanalysistools.ChordClass.extent_to_cardinality(7)
4
```

Returns integer.

`ChordClass.extent_to_extent_name` (*extent*)

Change *extent* to extent name.

**Example:** extent of seven is a seventh:

```
>>> tonalanalysistools.ChordClass.extent_to_extent_name(7)
'seventh'
```

Returns string.

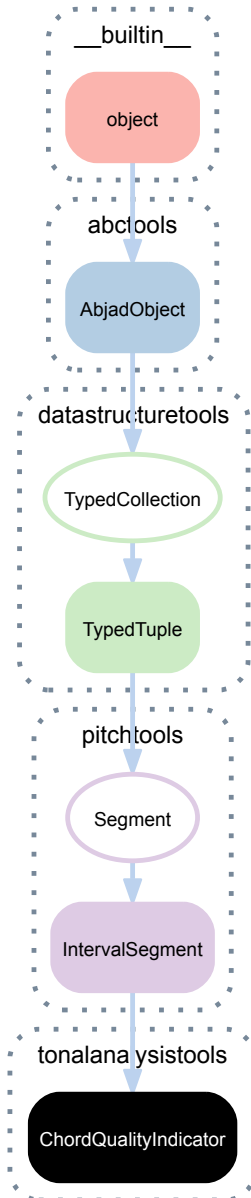
## Special methods

```
(TypedFrozenSet).__and__(expr)
(TypedCollection).__contains__(token)
ChordClass.__eq__(arg)
(TypedFrozenSet).__ge__(expr)
(TypedFrozenSet).__gt__(expr)
(PitchClassSet).__hash__()
(TypedCollection).__iter__()
(TypedFrozenSet).__le__(expr)
(TypedCollection).__len__()
(TypedFrozenSet).__lt__(expr)
ChordClass.__ne__(arg)
(TypedFrozenSet).__or__(expr)
ChordClass.__repr__()
(Set).__str__()
```

```
(TypedFrozenSet) .__sub__(expr)
```

```
(TypedFrozenSet) .__xor__(expr)
```

### 40.1.2 tonalanalysistools.ChordQualityIndicator



**class** `tonalanalysistools.ChordQualityIndicator` (*quality\_string*, *extent='triad'*, *inversion='root'*)

A chord quality indicator.

```
>>> tonalanalysistools.ChordQualityIndicator(
...     'German',
...     'augmented sixth',
... )
GermanAugmentedSixthInRootPosition('P1', '+M3', '+m3', '+aug2')
```

Returns chord quality indicator.

## Bases

- `pitchtools.IntervalSegment`
- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ChordQualityIndicator.cardinality`

`ChordQualityIndicator.extent`

`ChordQualityIndicator.extent_name`

`(IntervalSegment).has_duplicates`

True if segment contains duplicate items:

```
>>> intervals = 'm2 M3 -aug4 m2 P5'
>>> segment = pitchtools.IntervalSegment(intervals)
>>> segment.has_duplicates
True
```

```
>>> intervals = 'M3 -aug4 m2 P5'
>>> segment = pitchtools.IntervalSegment(intervals)
>>> segment.has_duplicates
False
```

Returns boolean.

`ChordQualityIndicator.inversion`

`(TypedCollection).item_class`

Item class to coerce tokens into.

`ChordQualityIndicator.position`

`ChordQualityIndicator.quality_string`

`ChordQualityIndicator.rotation`

`(IntervalSegment).slope`

The slope of a interval segment is the sum of its intervals divided by its length:

```
>>> pitchtools.IntervalSegment([1, 2]).slope
Multiplier(3, 2)
```

Returns multiplier.

`(IntervalSegment).spread`

The maximum interval spanned by any combination of the intervals within a numbered interval segment:

```
>>> pitchtools.IntervalSegment([1, 2, -3, 1, -2, 1]).spread
NumberedInterval(+4.0)
```

```
>>> pitchtools.IntervalSegment([1, 1, 1, 2, -3, -2]).spread
NumberedInterval(+5.0)
```

Returns numbered interval.

`(TypedCollection).storage_format`

Storage format of typed tuple.

## Read/write properties

(TypedCollection) **.name**  
Read / write name of typed tuple.

## Methods

(TypedTuple) **.count** (*token*)  
Change *token* to item and return count in collection.

(TypedTuple) **.index** (*token*)  
Change *token* to item and return index in collection.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

(IntervalSegment) **.rotate** (*n*)

## Class methods

(IntervalSegment) **.from\_selection** (*selection, item\_class=None, name=None*)  
Initialize interval segment from component selection:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> pitchtools.IntervalSegment.from_selection(
...     staff, item_class=pitchtools.NumberedInterval)
IntervalSegment([+2, +2, +1, +2, +2, +2, +1])
```

Returns interval segment.

## Static methods

ChordQualityIndicator **.from\_interval\_class\_segment** (*segment*)

## Special methods

(TypedTuple) **.\_\_add\_\_** (*expr*)

(TypedTuple) **.\_\_contains\_\_** (*token*)  
Change *token* to item and return true if item exists in collection.

(TypedCollection) **.\_\_eq\_\_** (*expr*)

(TypedTuple) **.\_\_getitem\_\_** (*i*)  
Aliases tuple.\_\_getitem\_\_().

(TypedTuple) **.\_\_getslice\_\_** (*start, stop*)

(TypedTuple) **.\_\_hash\_\_** ()

(TypedCollection) **.\_\_iter\_\_** ()

(TypedCollection) **.\_\_len\_\_** ()

(TypedTuple) **.\_\_mul\_\_** (*expr*)

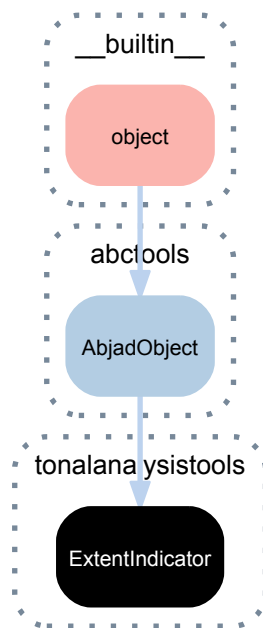
(TypedCollection) **.\_\_ne\_\_** (*expr*)

ChordQualityIndicator **.\_\_repr\_\_** ()

(TypedTuple) **.\_\_rmul\_\_** (*expr*)

(Segment) **.\_\_str\_\_** ()

### 40.1.3 tonalanalysistools.ExtentIndicator



**class** `tonalanalysistools.ExtentIndicator` (*arg*)  
A chord extent indicator, such as triad, seventh chord, ninth chord, etc.  
Value object that can not be changed after instantiation.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`ExtentIndicator.name`

`ExtentIndicator.number`

#### Special methods

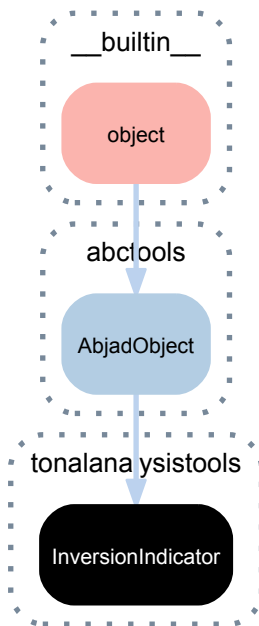
`ExtentIndicator.__eq__(arg)`

`ExtentIndicator.__ne__(arg)`

`ExtentIndicator.__repr__()`



#### 40.1.4 tonalanalysistools.InversionIndicator



**class** `tonalanalysistools.InversionIndicator` (*arg=0*)  
 An inversion indicator for tertian chords: 5, 63, 64 and also 7, 65, 43, 42, etc.  
 Also root position, first, second, third inversions, etc.  
 Value object that can not be changed once initialized.

##### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

##### Read-only properties

`InversionIndicator.name`  
`InversionIndicator.number`  
`InversionIndicator.title`

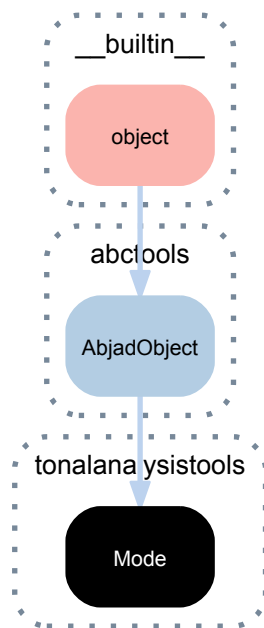
##### Methods

`InversionIndicator.extent_to_figured_bass_string` (*extent*)

##### Special methods

`InversionIndicator.__eq__` (*arg*)  
`InversionIndicator.__ne__` (*arg*)  
`InversionIndicator.__repr__` ()

### 40.1.5 tonalanalysistools.Mode



**class** `tonalanalysistools.Mode` (*arg*)

A diatonic mode.

Can be extended for nondiatonic mode.

Modes with different ascending and descending forms not yet implemented.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`Mode.mode_name`

`Mode.named_interval_segment`

#### Special methods

`Mode.__eq__` (*arg*)

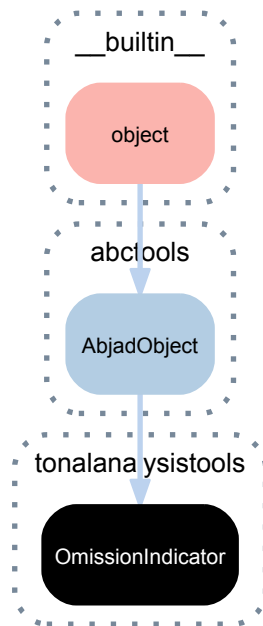
`Mode.__len__` ()

`Mode.__ne__` (*arg*)

`Mode.__repr__` ()

`Mode.__str__` ()

### 40.1.6 tonalanalysistools.OmissionIndicator



**class** `tonalanalysistools.OmissionIndicator`  
 An indicator of missing chord tones.  
 Value object that can not be chnaged after instantiation.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

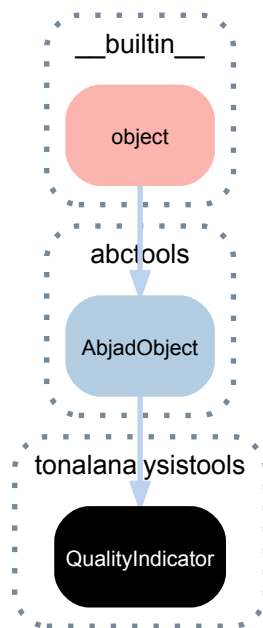
#### Special methods

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 40.1.7 tonalanalysistools.QualityIndicator



**class** `tonalanalysistools.QualityIndicator` (*quality\_string*)  
An indicator of chord quality, such as major, minor, dominant, diminished, etc.  
Value object that can not be changed after instantiation.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`QualityIndicator.is_uppercase`

`QualityIndicator.quality_string`

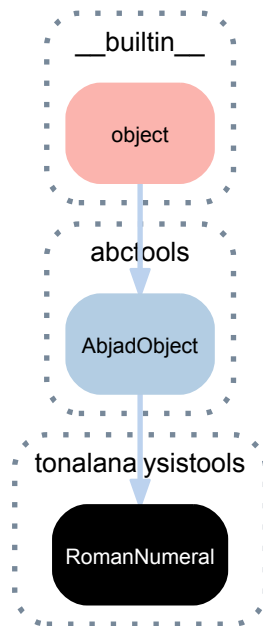
#### Special methods

`QualityIndicator.__eq__(arg)`

`QualityIndicator.__ne__(arg)`

`QualityIndicator.__repr__()`

### 40.1.8 tonalanalysistools.RomanNumeral



**class** `tonalanalysistools.RomanNumeral` (\*args)

A functions in tonal harmony: I, I6, I64, V, V7, V43, V42, bII, bII6, etc., also i, i6, i64, v, v7, etc.

Value object that can not be changed after instantiation.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`RomanNumeral.bass_scale_degree`

`RomanNumeral.extent`

`RomanNumeral.figured_bass_string`

`RomanNumeral.inversion`

`RomanNumeral.markup`

`RomanNumeral.quality`

`RomanNumeral.root_scale_degree`

`RomanNumeral.scale_degree`

`RomanNumeral.suspension`

`RomanNumeral.symbolic_string`

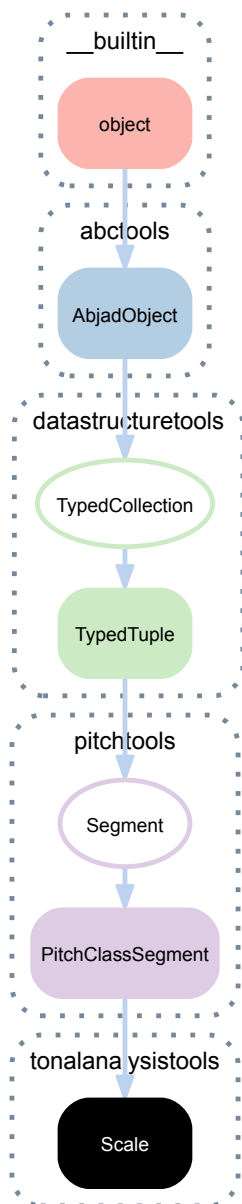
#### Special methods

`RomanNumeral.__eq__(arg)`

`RomanNumeral.__ne__(arg)`

`RomanNumeral.__repr__()`

### 40.1.9 tonalanalysistools.Scale



**class** `tonalanalysistools.Scale(*args)`  
A diatonic scale.

```
>>> scale = tonalanalysistools.Scale('C', 'minor')
```

#### Bases

- `pitchtools.PitchClassSegment`
- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

Scale.**dominant**

(PitchClassSegment).**has\_duplicates**

True if segment contains duplicate items:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.has_duplicates
True
```

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(
...     tokens="c d e f g a b",
...     )
>>> pitch_class_segment.has_duplicates
False
```

Returns boolean.

(TypedCollection).**item\_class**

Item class to coerce tokens into.

Scale.**key\_signature**

Scale.**leading\_tone**

Scale.**mediant**

Scale.**named\_interval\_class\_segment**

(TypedCollection).**storage\_format**

Storage format of typed tuple.

Scale.**subdominant**

Scale.**submediant**

Scale.**superdominant**

Scale.**tonic**

## Read/write properties

(TypedCollection).**name**

Read / write name of typed tuple.

## Methods

(PitchClassSegment).**alpha()**

Morris alpha transform of pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.alpha()
PitchClassSegment([11, 11.5, 7, 6, 11.5, 6])
```

Emit new pitch-class segment.

(TypedTuple).**count** (*token*)

Change *token* to item and return count in collection.

Scale.**create\_named\_pitch\_set\_in\_pitch\_range** (*pitch\_range*)

(TypedTuple).**index** (*token*)

Change *token* to item and return index in collection.

`(PitchClassSegment).invert()`

Invert pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.invert()
PitchClassSegment([2, 1.5, 6, 5, 1.5, 5])
```

Emit new pitch-class segment.

`(PitchClassSegment).is_equivalent_under_transposition(expr)`

True if equivalent under transposition to *expr*, otherwise False.

Returns boolean.

`Scale.make_notes(n, written_duration=None)`

Make first *n* notes in ascending diatonic scale. according to *key\_signature*.

Set *written\_duration* equal to *written\_duration* or 1/8:

```
>>> scale = tonalanalysistools.Scale('c', 'major')
>>> notes = scale.make_notes(8)
>>> staff = Staff(notes)
```

```
>>> show(staff)
```



Allow nonassignable *written\_duration*:

```
>>> notes = scale.make_notes(4, Duration(5, 16))
>>> staff = Staff(notes)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))(staff)
```

```
>>> show(staff)
```



Returns list of notes.

`Scale.make_score()`

Make MIDI playback score from scale:

```
>>> scale = tonalanalysistools.Scale('E', 'major')
>>> score = scale.make_score()
```

```
>>> show(score)
```



Returns score.

`(PitchClassSegment).multiply(n)`

Multiply pitch-class segment by *n*:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.multiply(5)
PitchClassSegment([2, 4.5, 6, 11, 4.5, 11])
```



Emit new pitch-class segment.

`Scale.named_pitch_class_to_scale_degree(*args)`

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

`(PitchClassSegment).retrograde()`

Retrograde of pitch-class segment:

```
>>> pitchtools.PitchClassSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).retrograde()
PitchClassSegment([7, 10.5, 7, 6, 10.5, 10])
```

Emit new pitch-class segment.

`(PitchClassSegment).rotate(n)`

Rotate pitch-class segment:

```
>>> pitchtools.PitchClassSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).rotate(1)
PitchClassSegment([7, 10, 10.5, 6, 7, 10.5])
```

```
>>> pitchtools.PitchClassSegment(
...     tokens=['c', 'ef', 'bqs', 'd'],
...     ).rotate(-2)
PitchClassSegment(['bqs', 'd', 'c', 'ef'])
```

Emit new pitch-class segment.

`Scale.scale_degree_to_named_pitch_class(*args)`

`(PitchClassSegment).transpose(expr)`

Transpose pitch-class segment:

```
>>> pitchtools.PitchClassSegment(
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],
...     ).transpose(10)
PitchClassSegment([8, 8.5, 4, 5, 8.5, 5])
```

Emit new pitch-class segment.

## Class methods

`Scale.from_selection(selection, item_class=None, name=None)`

## Special methods

`(TypedTuple).__add__(expr)`

`(TypedTuple).__contains__(token)`

Change *token* to item and return true if item exists in collection.

`(TypedCollection).__eq__(expr)`

`(TypedTuple).__getitem__(i)`

Aliases `tuple.__getitem__()`.

`(TypedTuple).__getslice__(start, stop)`

`(TypedTuple).__hash__()`

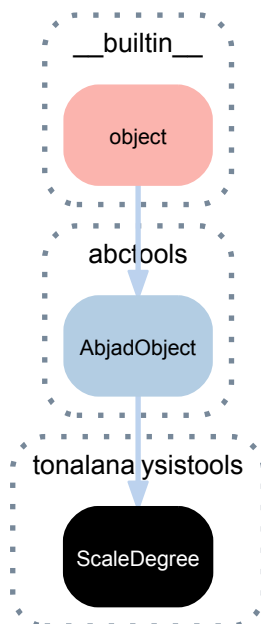
`(TypedCollection).__iter__()`

`(TypedCollection).__len__()`

`(TypedTuple).__mul__(expr)`

```
(TypedCollection).__ne__(expr)
Scale.__repr__()
(TypedTuple).__rmul__(expr)
(Segment).__str__()
```

#### 40.1.10 tonalanalysistools.ScaleDegree



```
class tonalanalysistools.ScaleDegree(*args)
    A diatonic scale degree such as 1, 2, 3, 4, 5, 6, 7 and also chromatic alterations including flat-2, flat-3, flat-6, etc.
```

##### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

##### Read-only properties

`ScaleDegree.accidental`  
Accidental applied to scale degree.

`ScaleDegree.name`  
Name of scale degree.

`ScaleDegree.number`  
Number of diatonic scale degree from 1 to 7, inclusive.

`ScaleDegree.roman_numeral_string`

`ScaleDegree.symbolic_string`

`ScaleDegree.title_string`

## Methods

`ScaleDegree.apply_accidental(accidental)`  
 Apply accidental to self and emit new instance.

## Special methods

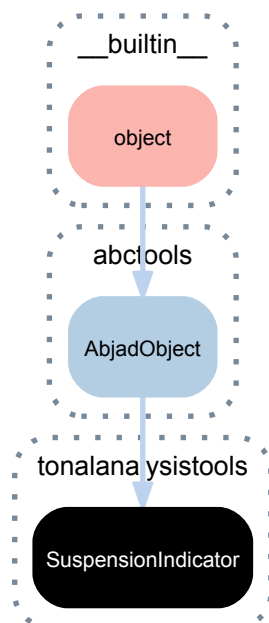
`ScaleDegree.__eq__(arg)`

`ScaleDegree.__ne__(arg)`

`ScaleDegree.__repr__()`

`ScaleDegree.__str__()`

### 40.1.11 tonalanalysistools.SuspensionIndicator



**class** `tonalanalysistools.SuspensionIndicator(*args)`  
 An indicator of 9-8, 7-6, 4-3, 2-1 and other types of suspension typical of, for example, the Bach chorales.  
 Value object that can not be changed after instantiation.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`SuspensionIndicator.chord_name`

`SuspensionIndicator.figured_bass_pair`

`SuspensionIndicator.figured_bass_string`

`SuspensionIndicator.is_empty`

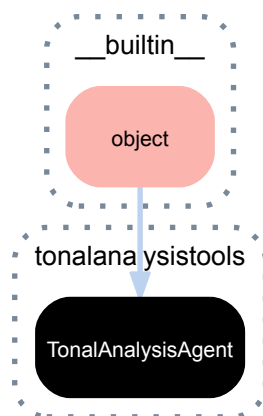
`SuspensionIndicator.start`

`SuspensionIndicator.stop`  
`SuspensionIndicator.title_string`

## Special methods

`SuspensionIndicator.__eq__(arg)`  
`SuspensionIndicator.__ne__(arg)`  
`SuspensionIndicator.__repr__()`  
`SuspensionIndicator.__str__()`

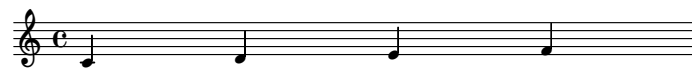
### 40.1.12 tonalanalysistools.TonalAnalysisAgent



**class** `tonalanalysistools.TonalAnalysisAgent` (*selection=None*)  
 A tonal analysis interface.

**Example 1.** Interface to conjunct selection:

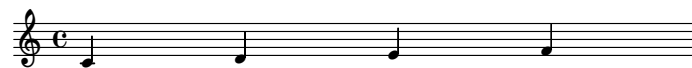
```
>>> staff = Staff("c'4 d' e' f'")
>>> show(staff)
```



```
>>> selection_1 = tonalanalysistools.select(staff[:])
```

**Example 2.** Interface to disjunct selection:

```
>>> staff = Staff("c'4 d' e' f'")
>>> show(staff)
```



```
>>> selection_2 = tonalanalysistools.select(staff[:1] + staff[-1:])
```

## Bases

- `__builtin__.object`

## Methods

`TonalAnalysisAgent.analyze_chords()`

Analyzes chords in selection.

```
>>> chord = Chord([7, 10, 12, 16], (1, 4))
>>> tonalanalysistools.select(chord).analyze_chords()
[CDominantSeventhInSecondInversion]
```

Returns none when no tonal chord is understood.

Returns list with elements each equal to chord class or none.

`TonalAnalysisAgent.analyze_incomplete_chords()`

Analyzes incomplete chords in selection.

```
>>> chord = Chord("<g' b'>4")
>>> tonalanalysistools.select(chord).analyze_incomplete_chords()
[GMajorTriadInRootPosition]
```

```
>>> chord = Chord("<fs g b>4")
>>> tonalanalysistools.select(chord).analyze_incomplete_chords()
[GMajorSeventhInSecondInversion]
```

Raises tonal harmony error when chord in selection can not analyze.

Returns list with elements each equal to chord class or none.

`TonalAnalysisAgent.analyze_incomplete_tonal_functions(key_signature)`

Analyzes incomplete tonal functions of chords in selection according to *key\_signature*.

```
>>> chord = Chord("<c' e'>4")
>>> key_signature = contextttools.KeySignatureMark('g', 'major')
>>> selection = tonalanalysistools.select(chord)
>>> selection.analyze_incomplete_tonal_functions(key_signature)
[IVMajorTriadInRootPosition]
```

Raises tonal harmony error when chord in selection can not analyze.

Returns list with elements each equal to tonal function or none.

`TonalAnalysisAgent.analyze_neighbor_notes()`

True when *note* in selection is preceeded by a stepwise interval in one direction and followed by a stepwise interval in the other direction. Otherwise false.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> selection = tonalanalysistools.select(staff[:])
>>> selection.analyze_neighbor_notes()
[False, False, False, False]
```

Returns list of boolean values.

`TonalAnalysisAgent.analyze_passing_tones()`

True when note in selection is both preceeded and followed by scalewise notes. Otherwise false.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> selection = tonalanalysistools.select(staff[:])
>>> selection.analyze_passing_tones()
[False, True, True, False]
```

Returns list of boolean values.

`TonalAnalysisAgent.analyze_tonal_functions(key_signature)`

Analyzes tonal function of chords in selection according to *key\_signature*.

```
>>> chord = Chord('<ef g bf>4')
>>> key_signature = contextttools.KeySignatureMark('c', 'major')
>>> selection = tonalanalysistools.select(chord)
>>> selection.analyze_tonal_functions(key_signature)
[FlatIIIMajorTriadInRootPosition]
```

Returns none when no tonal function is understood.

Returns list with elements each equal to tonal function or none.

`TonalAnalysisAgent.are_scalar_notes()`

True when notes in selection are scalar.

```
>>> selection_1.are_scalar_notes()
True
```

Otherwise false:

```
>>> selection_2.are_scalar_notes()
False
```

Returns boolean.

`TonalAnalysisAgent.are_stepwise_ascending_notes()`

True when notes in selection are stepwise ascending.

```
>>> selection_1.are_stepwise_ascending_notes()
True
```

Otherwise false:

```
>>> selection_2.are_stepwise_ascending_notes()
False
```

Returns boolean.

`TonalAnalysisAgent.are_stepwise_descending_notes()`

True when notes in selection are stepwise descending.

```
>>> selection_3 = tonalanalysistools.select(reversed(staff[:]))
```

```
>>> selection_3.are_stepwise_descending_notes()
True
```

Otherwise false:

```
>>> selection_1.are_stepwise_descending_notes()
False
```

```
>>> selection_2.are_stepwise_descending_notes()
False
```

Returns boolean.

`TonalAnalysisAgent.are_stepwise_notes()`

True when notes in selection are stepwise.

```
>>> selection_1.are_stepwise_notes()
True
```

Otherwise false:

```
>>> selection_2.are_stepwise_notes()
False
```

Returns boolean.

## Special methods

`TonalAnalysisAgent.__repr__()`

Interpreter representation of tonal analysis interface.

```
>>> selection_2
TonalAnalysisAgent(Note("c'4"), Note("f'4"))
```

Returns string.

## 40.2 Functions

### 40.2.1 tonalanalysistools.select

`tonalanalysistools.select` (*expr*)

Select *expr* for tonal analysis.

Returns tonal analysis selection.

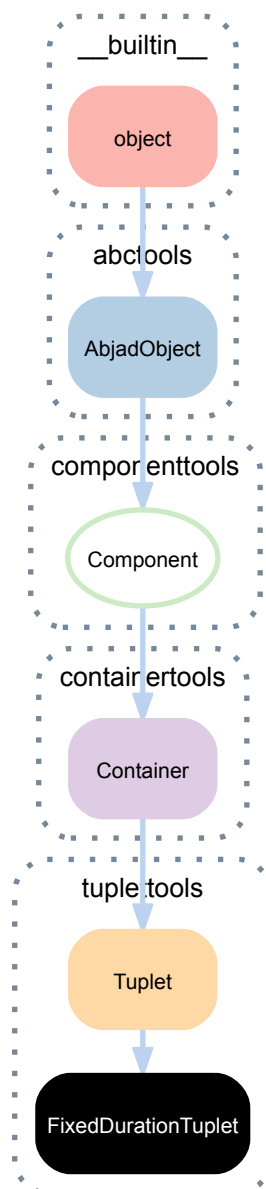




# TUPLETOOLS

## 41.1 Concrete classes

### 41.1.1 tupletools.FixedDurationTuplet



**class** tuplettools.**FixedDurationTuplet** (*duration, music=None, \*\*kwargs*)  
A tuplet with fixed duration and variable multiplier.

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), [])
>>> tuplet.extend("c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.append("fs'4")
>>> show(tuplet)
```



## Bases

- `tuplettools.Tuplet`
- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Tuplet).**implied\_prolation**  
Implied prolation of tuplet.

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.implied_prolation
Multiplier(2, 3)
```

Defined equal to tuplet multiplier.

Returns multiplier.

(Tuplet).**is\_augmentation**  
True when tuplet multiplier is greater than 1. Otherwise false.

**Example 1.** Augmented tuplet:

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
True
```

**Example 2.** Diminished tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
False
```

**Example 3. Trivial tuplet:**

```
>>> tuplet = Tuplet((1, 1), "c'8. d'8. e'8.")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
False
```

Returns boolean.

`(Tuplet).is_diminution`

True when tuplet multiplier is less than 1. Otherwise false.

**Example 1. Augmented tuplet:**

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
False
```

**Example 2. Diminished tuplet:**

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
True
```

**Example 3. Trivial tuplet:**

```
>>> tuplet = Tuplet((1, 1), "c'8. d'8. e'8.")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
False
```

Returns boolean.

`(Tuplet).is_trivial`

True when tuplet multiplier is equal to 1. Otherwise false:

```
>>> tuplet = Tuplet((1, 1), "c'8 d'8 e'8")
>>> tuplet.is_trivial
True
```

Returns boolean.

`(Tuplet).lilypond_format`

LilyPond format of tuplet.

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> print tuplet.lilypond_format
      times 2/3 {
        c'8
        d'8
        e'8
      }
```

Returns string.

`FixedDurationTuplet.multiplied_duration`

Multiplied duration of tuplet:

```
>>> tuplet = tuplettools.FixedDurationTuplet((1, 4), "c'8 d'8 e'8")
>>> tuplet.multiplied_duration
Duration(1, 4)
```

Returns duration.

`(Component).override`

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

`(Component).set`

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

`(Component).storage_format`

Storage format of component.

Returns string.

## Read/write properties

`(Tuplet).force_fraction`

Forced fraction formatting of tuplet.

**Example 1.** Get forced fraction formatting of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.force_fraction is None
True
```

**Example 2.** Set forced fraction formatting of tuplet:

```
>>> tuplet.force_fraction = True
>>> show(tuplet)
```



Returns boolean or none.

`(Tuplet).is_invisible`

Invisibility status of tuplet.

**Example.** Get tuplet invisibility status:

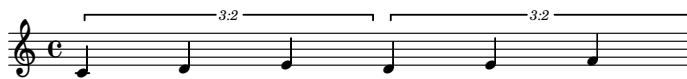
```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



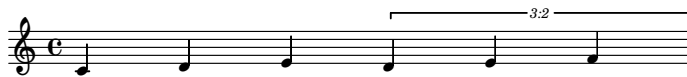
```
>>> tuplet.is_invisible is None
True
```

**Example 2.** Set tuplet invisibility status:

```
>>> tuplet_1 = Tuplet((2, 3), "c'4 d'4 e'4")
>>> tuplet_2 = Tuplet((2, 3), "d'4 e'4 f'4")
>>> staff = Staff([tuplet_1, tuplet_2])
>>> show(staff)
```



```
>>> staff[0].is_invisible = True
>>> show(staff)
```



Hides tuplet bracket and tuplet number when true.

Preserves tuplet duration when true.

Returns boolean or none.

`(Container).is_simultaneous`

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

`FixedDurationTuplet`.**`multiplier`**

Multiplier of tuplet:

```
>>> tuplet = tuplettools.FixedDurationTuplet(  
...     (1, 4), "c'8 d'8 e'8")  
>>> tuplet.multiplier  
Multiplier(2, 3)
```

Returns multiplier.

`(Tuplet)`.**`preferred_denominator`**

Preferred denominator of tuplet.

**Example 1.** Get preferred denominator of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")  
>>> tuplet.preferred_denominator is None  
True  
>>> show(tuplet)
```



**Example 2.** Set preferred denominator of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")  
>>> show(tuplet)
```



```
>>> tuplet.preferred_denominator = 4  
>>> show(tuplet)
```



Returns positive integer or none.

`FixedDurationTuplet`.**`target_duration`**

Read / write target duration of fixed-duration tuplet:

```
>>> tuplet = tuplettools.FixedDurationTuplet(  
...     (1, 4), "c'8 d'8 e'8")  
>>> tuplet.target_duration  
Duration(1, 4)
```

```
>>> tuplet.target_duration = Duration(5, 8)  
>>> f(tuplet)  
\tweak #'text #tuplet-number::calc-fraction-text  
\times 5/3 {  
    c'8  
    d'8  
    e'8  
}
```

Returns duration.

## Methods

(Container) **.append** (*component*)  
 Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) **.extend** (*expr*)  
 Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



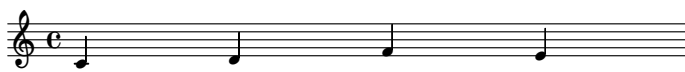
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) **.index** (*component*)  
 Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
```

```
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2. Insert note. Fracture spanners:**

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

`(Container).pop(i=-1)`

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

`(Container).remove(component)`

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```





```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container).**reverse**()  
Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component).**.select** (*sequential=False*)  
Selects component.  
  
Returns component selection when *sequential* is false.  
Returns sequential selection when *sequential* is true.

(Container).**.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)  
Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest("r8"), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container).**.select\_notes\_and\_chords**()  
Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

(Tuplet).**.set\_minimum\_denominator** (*denominator*)  
Sets preferred denominator of tuplet to at least *denominator*.

**Example.** Set preferred denominator of tuplet to at least 8:

```
>>> tuplet = Tuplet((3, 5), "c'4 d'8 e'8 f'4 g'2")
>>> show(tuplet)
```



```
>>> tuplet.set_minimum_denominator(8)
>>> show(tuplet)
```



Returns none.

`(Tuplet).to_fixed_duration_tuplet()`

Change tuplet to fixed-duration tuplet.

**Example:**

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet
Tuplet(2/3, [c'8, d'8, e'8])
```

```
>>> new_tuplet = tuplet.to_fixed_duration_tuplet()
>>> show(new_tuplet)
```



```
>>> new_tuplet
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

Returns new tuplet.

`FixedDurationTuplet.to_fixed_multiplier()`

Change fixed-duration tuplet to (unqualified) tuplet.

**Example:**

```
>>> tuplet = tuplettools.FixedDurationTuplet((2, 8), [])
>>> tuplet.extend("c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

```
>>> new_tuplet = tuplet.to_fixed_multiplier()
>>> show(new_tuplet)
```



```
>>> new_tuplet
Tuplet(2/3, [c'8, d'8, e'8])
```

Returns new tuplet.

`FixedDurationTuplet.toggle_prolation()`

`FixedDurationTuplet.trim(start, stop='unused')`

Trim fixed-duration tuplet elements from *start* to *stop*:

```
>>> tuplet = tuplettools.FixedDurationTuplet(
...     Fraction(2, 8), "c'8 d'8 e'8")
>>> tuplet
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

```
>>> tuplet.trim(2)
>>> tuplet
FixedDurationTuplet(1/6, [c'8, d'8])
```

Preserve fixed-duration tuplet multiplier.

Adjust fixed-duration tuplet duration.

Returns none.

## Static methods

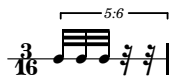
(Tuplet).**from\_duration\_and\_ratio**(*duration*, *ratio*, *avoid\_dots=True*, *decrease\_durations\_monotonically=True*, *is\_diminution=True*)

Makes tuplet from *duration* and *ratio*.

**Example 1.** Make augmented tuplet from *duration* and *ratio* and avoid dots.

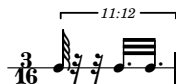
Make tupletted leaves strictly without dots when all *ratio* equal 1:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=True,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



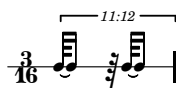
Allow tupletted leaves to return with dots when some *ratio* do not equal 1:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, -2, -2, 3, 3),
...     avoid_dots=True,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



Interpret nonassignable *ratio* according to *decrease\_durations\_monotonically*:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=True,
...     decrease_durations_monotonically=False,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



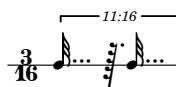
**Example 2.** Make augmented tuplet from *duration* and *ratio* and encourage dots:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=False,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



Interpret nonassignable *ratio* according to *decrease\_durations\_monotonically*:

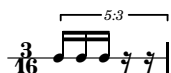
```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=False,
...     decrease_durations_monotonically=False,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



**Example 3.** Make diminished tuplet from *duration* and nonzero integer *ratio*.

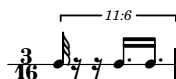
Make tupletted leaves strictly without dots when all *ratio* equal 1:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=True,
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



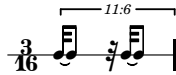
Allow tupletted leaves to return with dots when some *ratio* do not equal 1:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, -2, -2, 3, 3),
...     avoid_dots=True,
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



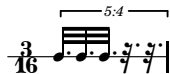
Interpret nonassignable *ratio* according to *decrease\_durations\_monotonically*:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=True,
...     decrease_durations_monotonically=False,
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



**Example 4.** Make diminished tuplet from *duration* and *ratio* and encourage dots:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=False,
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



Interpret nonassignable *ratio* according to *direction*:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=False,
...     decrease_durations_monotonically=False,
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(measure)
```



Reduces *ratio* relative to each other.

Interprets negative *ratio* as rests.

Returns fixed-duration tuplet.

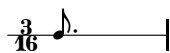
(Tuplet).**from\_leaf\_and\_ratio**(*leaf*, *ratio*, *is\_diminution*=True)

Makes tuplet from *leaf* and *ratio*.

```
>>> note = Note("c'8.")
```

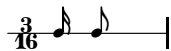
**Example 1a.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```



**Example 1b.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```



**Example 1c.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```



**Example 1d.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```



**Example 1e.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3, 3],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```



**Example 1f.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2, 3, 3, 4),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```



**Example 2a.** Change leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
```

```

...     [1],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))

```

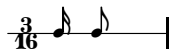


**Example 2b.** Change leaf to diminished tuplets with *ratio*:

```

>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))

```

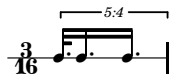


**Example 2c.** Change leaf to diminished tuplets with *ratio*:

```

>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))

```

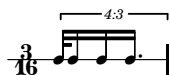


**Example 2d.** Change leaf to diminished tuplets with *ratio*:

```

>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))

```

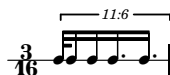


**Example 2e.** Change leaf to diminished tuplets with *ratio*:

```

>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3, 3],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))

```

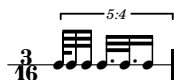


**Example 2f.** Change leaf to diminished tuplets with *ratio*:

```

>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3, 3, 4],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))

```



Returns tuplet.

`(Tuplet).from_nonreduced_ratio_and_nonreduced_fraction(ratio, fraction)`

Makes tuplet from nonreduced *ratio* and nonreduced *fraction*.

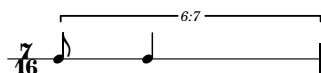
**Example 1.** Make container when no prolotion is necessary:

```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```

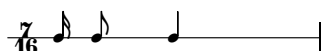


**Example 2.** Make fixed-duration tuplet when prolotion is necessary:

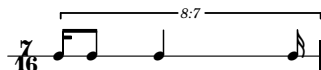
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



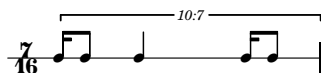
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```

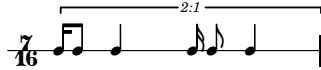


```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1, 2),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```





```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1, 2, 4),
...     mathtools.NonreducedFraction(7, 16),
...     )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



Interprets  $d$  as tuplet denominator.

Returns tuplet or container.

## Special methods

(Container).**\_\_contains\_\_**(*expr*)

True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component).**\_\_copy\_\_**(\*args)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(Container).**\_\_delitem\_\_**(*i*)

Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Returns none.

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container).**\_\_getitem\_\_**(*i*)

Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.

(Container).**\_\_len\_\_**()

Number of items in container.

Returns nonnegative integer.

(Component).**\_\_mul\_\_**(*n*)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

FixedDurationTuplet.**\_\_repr\_\_**()

Interpreter representation of fixd-duration tuplet.

Returns string.

(Component).**\_\_rmul\_\_**(*n*)

Copies component *n* times and detach spanners.

Returns list of new components.

(Container).**\_\_setitem\_\_**(*i, expr*)

Set container *i* equal to *expr*. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

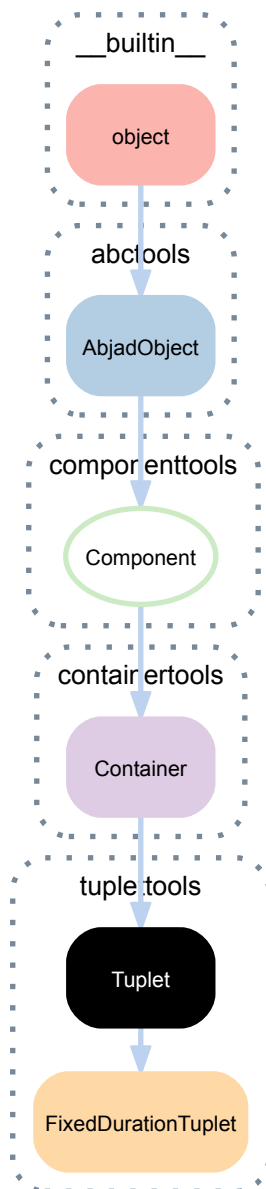
Returns none.

FixedDurationTuplet.**\_\_str\_\_**()

String representation of fixed-duration tuplet.

Returns string.

### 41.1.2 tuplettools.Tuplet



**class** tuplettools.**Tuplet** (*multiplier, music=None, \*\*kwargs*)

A tuplet.

**Example 1.** A tuplet:

```
>>> tuplet = Tuplet(Multiplier(2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```

**Example 2.** A nested tuplet:

```
>>> second_tuplet = Tuplet((4, 7), "g'4. ( a'16 )")
>>> tuplet.insert(1, second_tuplet)
>>> show(tuplet)
```

**Example 3.** A doubly nested tuplet:

```
>>> third_tuplet = Tuplet((4, 5), [])
>>> third_tuplet.extend("e''32 [ ef''32 d''32 cs''32 cqs''32 ]")
>>> second_tuplet.insert(1, third_tuplet)
>>> show(tuplet)
```



## Bases

- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `Tuplet.implied_prolation`

Implied prololation of tuplet.

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.implied_prolation
Multiplier(2, 3)
```

Defined equal to tuplet multiplier.

Returns multiplier.

### `Tuplet.is_augmentation`

True when tuplet multiplier is greater than 1. Otherwise false.

**Example 1.** Augmented tuplet:

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
True
```

**Example 2.** Diminished tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
False
```

**Example 3.** Trivial tuplet:

```
>>> tuplet = Tuplet((1, 1), "c'8. d'8. e'8.")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
False
```

Returns boolean.

**Tuplet.is\_diminution**

True when tuplet multiplier is less than 1. Otherwise false.

**Example 1.** Augmented tuplet:

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
False
```

**Example 2.** Diminished tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
True
```

**Example 3.** Trivial tuplet:

```
>>> tuplet = Tuplet((1, 1), "c'8. d'8. e'8.")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
False
```

Returns boolean.

**Tuplet.is\_trivial**

True when tuplet multiplier is equal to 1. Otherwise false:

```
>>> tuplet = Tuplet((1, 1), "c'8 d'8 e'8")
>>> tuplet.is_trivial
True
```

Returns boolean.

**Tuplet.lilypond\_format**

LilyPond format of tuplet.

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> print tuplet.lilypond_format
      imes 2/3 {
        c'8
        d'8
        e'8
      }
```

Returns string.

**Tuplet.multiplied\_duration**

Multiplied duration of tuplet.

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.multiplied_duration
Duration(1, 4)
```

Returns duration.

(Component) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Component) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

(Component) **.storage\_format**

Storage format of component.

Returns string.

## Read/write properties

**Tuplet.force\_fraction**

Forced fraction formatting of tuplet.

**Example 1.** Get forced fraction formatting of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.force_fraction is None
True
```

**Example 2.** Set forced fraction formatting of tuplet:

```
>>> tuplet.force_fraction = True
>>> show(tuplet)
```



Returns boolean or none.

`Tuplet.is_invisible`  
Invisibility status of tuplet.

**Example.** Get tuplet invisibility status:

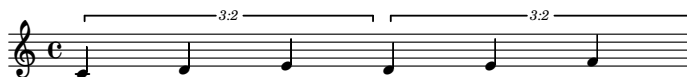
```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



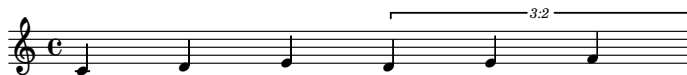
```
>>> tuplet.is_invisible is None
True
```

**Example 2.** Set tuplet invisibility status:

```
>>> tuplet_1 = Tuplet((2, 3), "c'4 d'4 e'4")
>>> tuplet_2 = Tuplet((2, 3), "d'4 e'4 f'4")
>>> staff = Staff([tuplet_1, tuplet_2])
>>> show(staff)
```



```
>>> staff[0].is_invisible = True
>>> show(staff)
```



Hides tuplet bracket and tuplet number when true.

Preserves tuplet duration when true.

Returns boolean or none.

`(Container).is_simultaneous`  
Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

### `Tuplet.multiplier`

Tuplet multiplier.

**Example 1.** Get tuplet multiplier:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.multiplier
Multiplier(2, 3)
```

**Example 2.** Set tuplet multiplier:

```
>>> tuplet.multiplier = Multiplier(4, 3)
>>> show(tuplet)
```



Returns multiplier.

### `Tuplet.preferred_denominator`

Preferred denominator of tuplet.

**Example 1.** Get preferred denominator of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.preferred_denominator is None
True
>>> show(tuplet)
```



**Example 2.** Set preferred denominator of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.preferred_denominator = 4
>>> show(tuplet)
```



Returns positive integer or none.

## Methods

(Container) .**append**(*component*)  
 Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



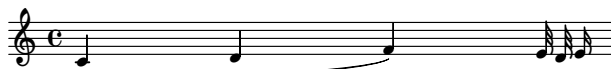
Returns none.

(Container) .**extend**(*expr*)  
 Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



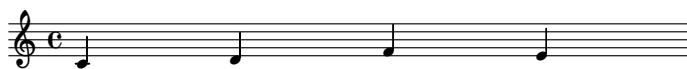
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

(Container) .**index**(*component*)  
 Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) .**insert**(*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
```



```
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2. Insert note. Fracture spanners:**

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

`(Container) .pop (i=-1)`

Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

`(Container) .remove (component)`

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse()**  
Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (*sequential=False*)  
Selects component.  
Returns component selection when *sequential* is false.  
Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (*start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False*)  
Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest("r8"), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords()**  
Selects notes and chords in container.

```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

Tuplet **.set\_minimum\_denominator** (*denominator*)  
Sets preferred denominator of tuplet to at least *denominator*.

**Example.** Set preferred denominator of tuplet to at least 8:

```
>>> tuplet = Tuplet((3, 5), "c'4 d'8 e'8 f'4 g'2")
>>> show(tuplet)
```



```
>>> tuplet.set_minimum_denominator(8)
>>> show(tuplet)
```



Returns none.

`Tuplet.to_fixed_duration_tuplet()`  
Change tuplet to fixed-duration tuplet.

**Example:**

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet
Tuplet(2/3, [c'8, d'8, e'8])
```

```
>>> new_tuplet = tuplet.to_fixed_duration_tuplet()
>>> show(new_tuplet)
```



```
>>> new_tuplet
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

Returns new tuplet.

`Tuplet.toggle_prolation()`  
Changes augmented tuplets to diminished; changes diminished tuplets to augmented.

**Example 1.** Change augmented tuplet to diminished:

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.toggle_prolation()
>>> show(tuplet)
```



Multiplies the written duration of the leaves in tuplet by the least power of 2 necessary to diminish tuplet.

**Example 2.** Change diminished tuplet to augmented:

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.toggle_prolation()
>>> show(tuplet)
```



Divides the written duration of the leaves in tuplet by the least power of 2 necessary to diminished tuplet.

Does not yet work with nested tuplets.

Returns none.

## Static methods

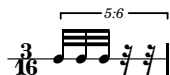
`Tuplet.from_duration_and_ratio` (*duration*, *ratio*, *avoid\_dots=True*, *decrease\_durations\_monotonically=True*, *is\_diminution=True*)

Makes tuplet from *duration* and *ratio*.

**Example 1.** Make augmented tuplet from *duration* and *ratio* and avoid dots.

Make tupletted leaves strictly without dots when all *ratio* equal 1:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=True,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



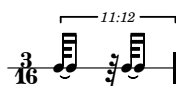
Allow tupletted leaves to return with dots when some *ratio* do not equal 1:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, -2, -2, 3, 3),
...     avoid_dots=True,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



Interpret nonassignable *ratio* according to *decrease\_durations\_monotonically*:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=True,
...     decrease_durations_monotonically=False,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



**Example 2.** Make augmented tuplet from *duration* and *ratio* and encourage dots:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=False,
```

```

...     is_diminution=False,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)

```

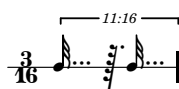


Interpret nonassignable *ratio* according to *decrease\_durations\_monotonically*:

```

>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=False,
...     decrease_durations_monotonically=False,
...     is_diminution=False,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)

```



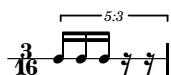
**Example 3.** Make diminished tuplet from *duration* and nonzero integer *ratio*.

Make tupletted leaves strictly without dots when all *ratio* equal 1:

```

>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=True,
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)

```

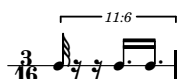


Allow tupletted leaves to return with dots when some *ratio* do not equal 1:

```

>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, -2, -2, 3, 3),
...     avoid_dots=True,
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)

```



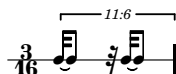
Interpret nonassignable *ratio* according to *decrease\_durations\_monotonically*:

```

>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=True,
...     decrease_durations_monotonically=False,
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])

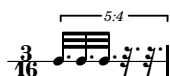
```

```
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



**Example 4.** Make diminished tuplet from *duration* and *ratio* and encourage dots:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=False,
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



Interpret nonassignable *ratio* according to *direction*:

```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=False,
...     decrease_durations_monotonically=False,
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(measure)
```



Reduces *ratio* relative to each other.

Interprets negative *ratio* as rests.

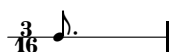
Returns fixed-duration tuplet.

`Tuplet.from_leaf_and_ratio(leaf, ratio, is_diminution=True)`  
 Makes tuplet from *leaf* and *ratio*.

```
>>> note = Note("c' 8.")
```

**Example 1a.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

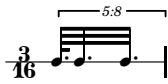


**Example 1b.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

**Example 1c.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

**Example 1d.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

**Example 1e.** Change leaf to augmented tuplets with *ratio*:

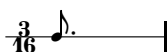
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3, 3],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

**Example 1f.** Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2, 3, 3, 4),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

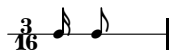
**Example 2a.** Change leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

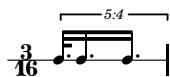


**Example 2b.** Change leaf to diminished tuplets with *ratio*:

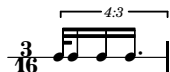
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

**Example 2c.** Change leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

**Example 2d.** Change leaf to diminished tuplets with *ratio*:

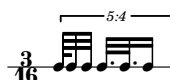
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

**Example 2e.** Change leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3, 3],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```

**Example 2f.** Change leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3, 3, 4],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(stafftools.RhythmicStaff([measure]))
```



Returns tuplet.

`Tuplet.from_nonreduced_ratio_and_nonreduced_fraction` (*ratio*, *fraction*)  
 Makes tuplet from nonreduced *ratio* and nonreduced *fraction*.



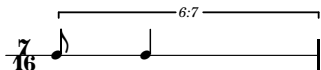
**Example 1.** Make container when no prolotion is necessary:

```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```

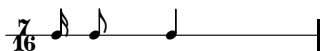


**Example 2.** Make fixed-duration tuplet when prolotion is necessary:

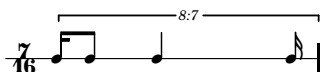
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



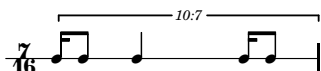
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



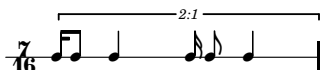
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1, 2),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1, 2, 4),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
>>> show(staff)
```



Interprets *d* as tuplet denominator.

Returns tuplet or container.

## Special methods

(Container).**\_\_contains\_\_**(*expr*)

True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component).**\_\_copy\_\_**(\*args)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(Container).**\_\_delitem\_\_**(*i*)

Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners.

Preserve spanners that component(s) cover(s).

Returns none.

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container).**\_\_getitem\_\_**(*i*)

Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.

(Container).**\_\_len\_\_**()

Number of items in container.

Returns nonnegative integer.

(Component).**\_\_mul\_\_**(*n*)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

Tuplet.**\_\_repr\_\_**()

Interpreter representation of tuplet.

Returns string.

(Component).**\_\_rmul\_\_**(*n*)

Copies component *n* times and detach spanners.

Returns list of new components.

(Container).**\_\_setitem\_\_**(*i*, *expr*)

Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with '*expr*'. Reattach spanners to new contents. This operation always leaves score tree in tact.

Returns none.

Tuplet.**\_\_str\_\_**()

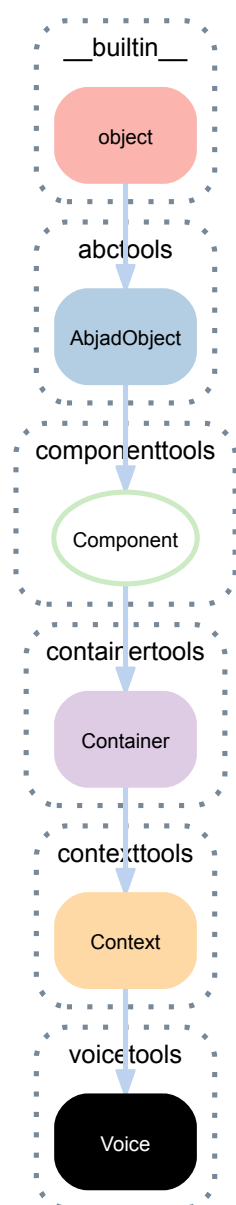
String representation of tuplet.

Returns string.

# VOICETOOLS

## 42.1 Concrete classes

### 42.1.1 voicetools.Voice



**class** `voicetools.Voice` (*music=None, context\_name='Voice', name=None*)  
 A musical voice.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
```

```
>>> voice
Voice{4}
```

```
>>> show(voice)
```



Returns voice instance.

## Bases

- `contexttools.Context`
- `containertools.Container`
- `componenttools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Context) **.engraver\_consists**

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

(Context) **.engraver\_removals**

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

(Context) **.is\_semantic**

(Context) **.lilypond\_format**

(Component) **.override**

LilyPond grob override component plug-in.

Returns LilyPond grob override component plug-in.

(Component) **.set**

LilyPond context setting component plug-in.

Returns LilyPond context setting component plug-in.

(Component).**storage\_format**

Storage format of component.

Returns string.

## Read/write properties

(Context).**context\_name**

Read / write name of context as a string.

(Context).**is\_nonsemantic**

Set indicator of nonsemantic voice:

```
>>> measures = \
...     measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Returns boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interaction and other functions.

(Container).**is\_simultaneous**

Simultaneity status of container.

**Example 1.** Get simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous
False
```

**Example 2.** Set simultaneity status of container:

```
>>> container = Container()
>>> container.append(Voice("c'8 d'8 e'8"))
>>> container.append(Voice('g4.'))
>>> show(container)
```



```
>>> container.is_simultaneous = True
>>> show(container)
```



Returns boolean.

(Context) **.name**

Read-write name of context. Must be string or none.

## Methods

(Container) **.append** (*component*)

Appends *component* to container.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> container.append(Note("e'4"))
>>> show(container)
```



Returns none.

(Container) **.extend** (*expr*)

Extends container with *expr*.

```
>>> container = Container("c'4 ( d'4 f'4 )")
>>> show(container)
```



```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```

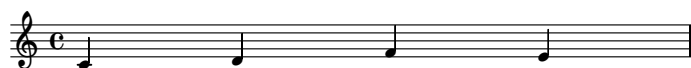


Returns none.

(Container) **.index** (*component*)

Returns index of *component* in container.

```
>>> container = Container("c'4 d'4 f'4 e'4")
>>> show(container)
```



```
>>> note = container[-1]
>>> note
Note("e'4")
```

```
>>> container.index(note)
3
```

Returns nonnegative integer.

(Container) **.insert** (*i*, *component*, *fracture\_spanners=False*)  
 Inserts *component* at index *i* in container.

**Example 1.** Insert note. Do not fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=False)
>>> show(container)
```



**Example 2.** Insert note. Fracture spanners:

```
>>> container = Container([])
>>> container.extend("fs16 cs' e' a'")
>>> container.extend("cs''16 e'' cs'' a'")
>>> container.extend("fs'16 e' cs' fs")
>>> slur = spannertools.SlurSpanner(container[:])
>>> slur.direction = Down
>>> show(container)
```



```
>>> container.insert(-4, Note("e'4"), fracture_spanners=True)
>>> show(container)
```



Returns none.

(Container) **.pop** (*i=-1*)  
 Pops component from container at index *i*.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> container.pop()
Note("e'4")
>>> show(container)
```



Returns component.

(Container) **.remove** (component)

Removes *component* from container.

```
>>> container = Container("c'4 ( d'4 f'4 ) e'4")
>>> show(container)
```



```
>>> note = container[2]
>>> note
Note("f'4")
```

```
>>> container.remove(note)
>>> show(container)
```



Returns none.

(Container) **.reverse** ()

Reverses contents of container.

```
>>> staff = Staff("c'8 [ d'8 ] e'8 ( f'8 )")
>>> show(staff)
```



```
>>> staff.reverse()
>>> show(staff)
```



Returns none.

(Component) **.select** (sequential=False)

Selects component.

Returns component selection when *sequential* is false.

Returns sequential selection when *sequential* is true.

(Container) **.select\_leaves** (start=0, stop=None, leaf\_classes=None, recurse=True, allow\_discontiguous\_leaves=False)

Selects leaves in container.

```
>>> container = Container("c'8 d'8 r8 e'8")
```

```
>>> container.select_leaves()
ContiguousSelection(Note("c'8"), Note("d'8"), Rest('r8'), Note("e'8"))
```

Returns contiguous leaf selection or free leaf selection.

(Container) **.select\_notes\_and\_chords** ()

Selects notes and chords in container.



```
>>> container.select_notes_and_chords()
Selection(Note("c'8"), Note("d'8"), Note("e'8"))
```

Returns leaf selection.

## Special methods

(Container) **.\_\_contains\_\_** (*expr*)

True when *expr* appears in container. Otherwise false.

Returns boolean.

(Component) **.\_\_copy\_\_** (\*args)

Copies component with marks but without children of component or spanners attached to component.

Returns new component.

(Container) **.\_\_delitem\_\_** (*i*)

Delete container *i*. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Returns none.

(AbjadObject) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(Container) **.\_\_getitem\_\_** (*i*)

Get container *i*. Shallow traversal of container for numeric indices only.

Returns component.

(Container) **.\_\_len\_\_** ()

Number of items in container.

Returns nonnegative integer.

(Component) **.\_\_mul\_\_** (*n*)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(Context) **.\_\_repr\_\_** ()

(Component) **.\_\_rmul\_\_** (*n*)

Copies component *n* times and detach spanners.

Returns list of new components.

(Container) **.\_\_setitem\_\_** (*i*, *expr*)

Set container *i* equal to *expr*. Find spanners that dominate self[*i*] and children of self[*i*]. Replace contents at self[*i*] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

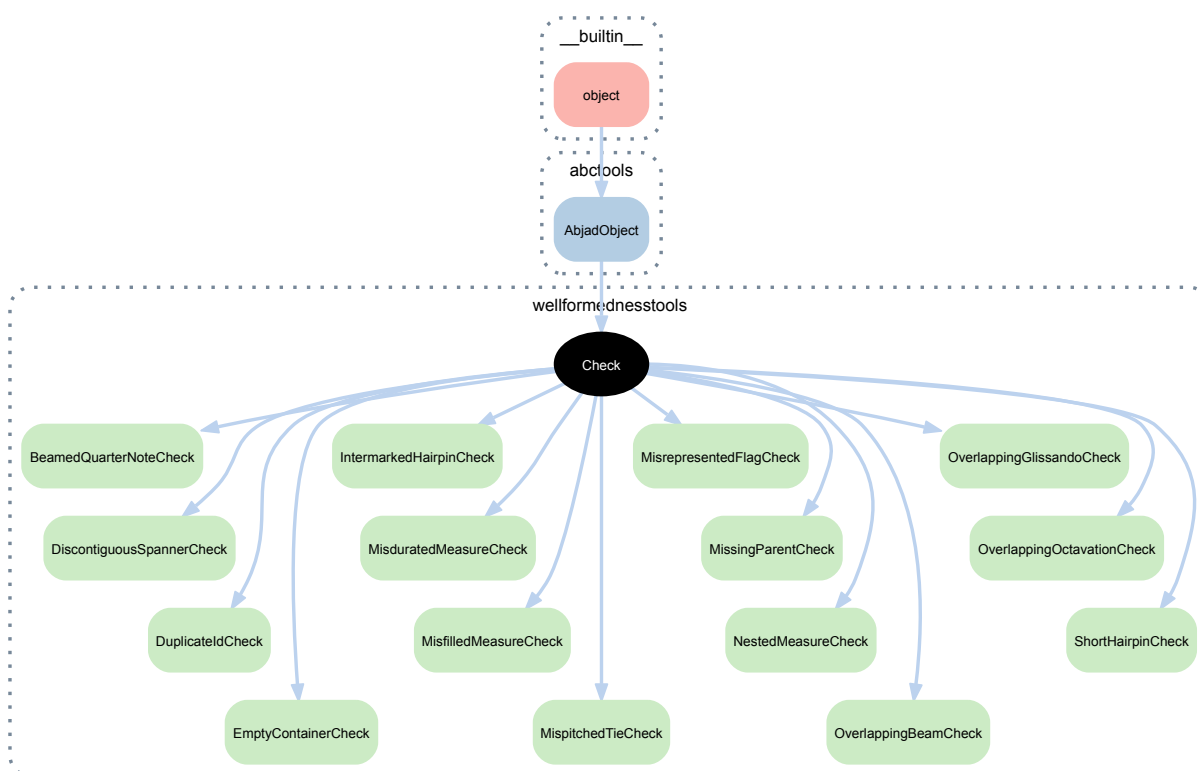
Returns none.



## WELLFORMEDNESSTOOLS

### 43.1 Abstract classes

#### 43.1.1 wellformednesstools.Check



**class** `wellformednesstools.Check`

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Methods

`Check.check` (*expr*)

`Check.report` (*expr*)

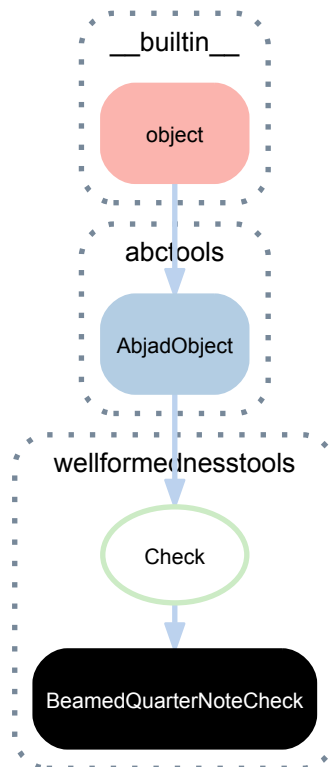
`Check.violators` (*expr*)

## Special methods

- (`AbjadObject`).`__eq__`(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.
- (`AbjadObject`).`__ne__`(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.
- (`AbjadObject`).`__repr__`()  
 Interpreter representation of Abjad object.  
 Returns string.

## 43.2 Concrete classes

### 43.2.1 `wellformednesstools.BeamedQuarterNoteCheck`



`class wellformednesstools.BeamedQuarterNoteCheck`

## Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

(Check) **.check** (*expr*)  
 (Check) **.report** (*expr*)  
 (Check) **.violators** (*expr*)

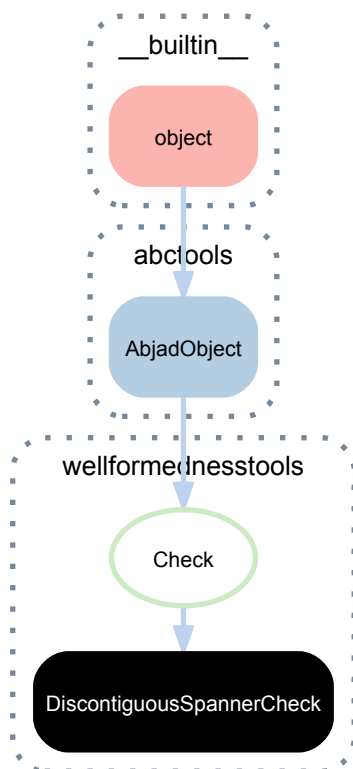
## Special methods

(AbjadObject) **.\_\_eq\_\_** (*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_repr\_\_** ()  
 Interpreter representation of Abjad object.  
 Returns string.

### 43.2.2 wellformednesstools.DiscontiguousSpannerCheck



#### class wellformednesstools.DiscontiguousSpannerCheck

There are now two different types of spanner. Most spanners demand that spanner components be logical-voice-contiguous. But a few special spanners (like Tempo) do not make such a demand. The check here consults the experimental `_contiguity_constraint`.

## Bases

- `wellformednesstools.Check`

- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

(Check) **.check** (*expr*)

(Check) **.report** (*expr*)

(Check) **.violators** (*expr*)

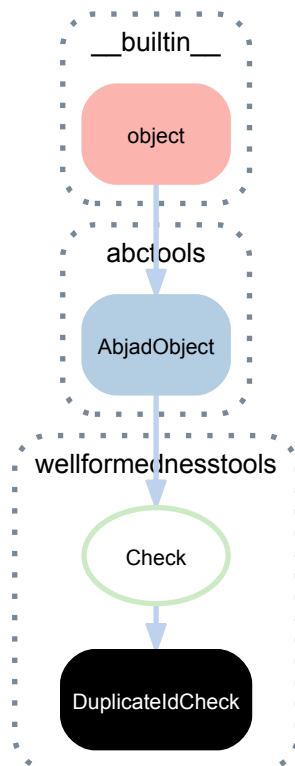
## Special methods

(AbjadObject) **.\_\_eq\_\_** (*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_repr\_\_** ()  
 Interpreter representation of Abjad object.  
 Returns string.

### 43.2.3 wellformednesstools.DuplicateIdCheck



**class** `wellformednesstools.DuplicateIdCheck`

## Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

`(Check).check(expr)`

`(Check).report(expr)`

`(Check).violators(expr)`

## Special methods

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

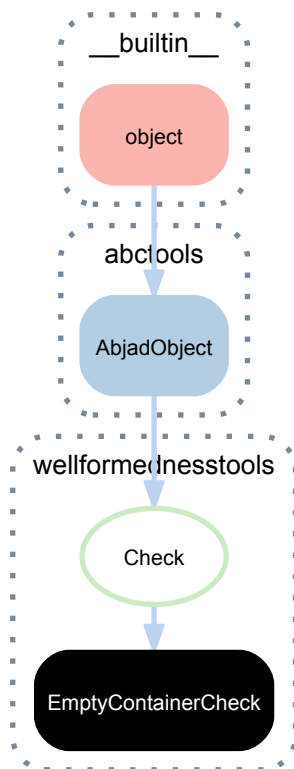
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 43.2.4 `wellformednesstools.EmptyContainerCheck`



`class wellformednesstools.EmptyContainerCheck`

## Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

(Check) **.check** (*expr*)

(Check) **.report** (*expr*)

(Check) **.violators** (*expr*)

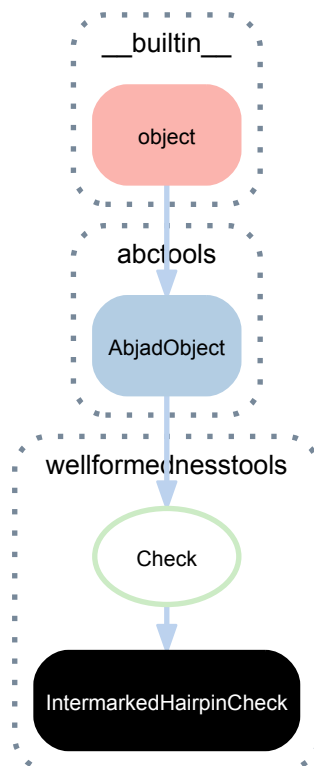
## Special methods

(AbjadObject) **.\_\_eq\_\_** (*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_repr\_\_** ()  
 Interpreter representation of Abjad object.  
 Returns string.

### 43.2.5 wellformednesstools.IntermarkedHairpinCheck





**class** `wellformednesstools.IntermarkedHairpinCheck`  
Are there any dynamic marks in the middle of a hairpin?

### Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

### Methods

(`Check`) **.check** (*expr*)

(`Check`) **.report** (*expr*)

(`Check`) **.violators** (*expr*)

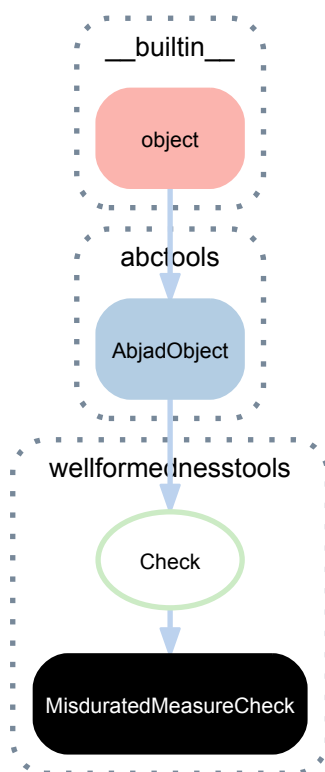
### Special methods

(`AbjadObject`) **.\_\_eq\_\_** (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(`AbjadObject`) **.\_\_ne\_\_** (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(`AbjadObject`) **.\_\_repr\_\_** ()  
Interpreter representation of Abjad object.  
Returns string.

### 43.2.6 wellformednesstools.MisduratedMeasureCheck



**class** wellformednesstools.MisduratedMeasureCheck

Does the (pre)rolated duration of the measure match its time signature?

#### Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Methods

(Check) **.check** (*expr*)

(Check) **.report** (*expr*)

(Check) **.violators** (*expr*)

#### Special methods

(AbjadObject) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

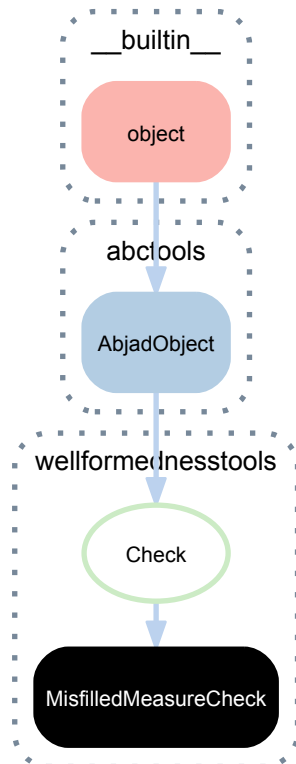
Returns boolean.

(AbjadObject) **.\_\_repr\_\_** ()

Interpreter representation of Abjad object.

Returns string.

### 43.2.7 wellformednesstools.MisfilledMeasureCheck



**class** `wellformednesstools.MisfilledMeasureCheck`

Check that time signature duration equals measure contents duration for every measure.

#### Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Methods

(`Check`) **.check** (*expr*)

(`Check`) **.report** (*expr*)

(`Check`) **.violators** (*expr*)

#### Special methods

(`AbjadObject`) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

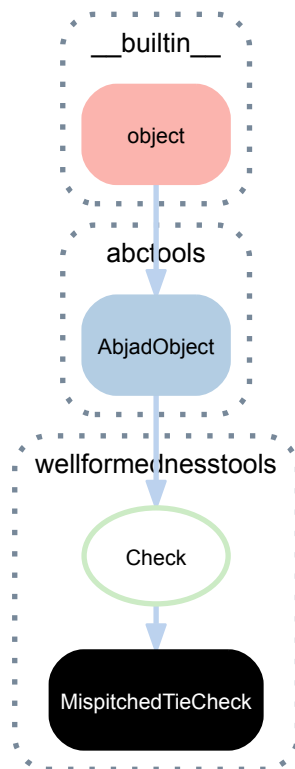
(`AbjadObject`) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 43.2.8 wellformednesstools.MispitchedTieCheck



**class** `wellformednesstools.MispitchedTieCheck`

#### Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Methods

`(Check).check(expr)`  
`(Check).report(expr)`  
`(Check).violators(expr)`

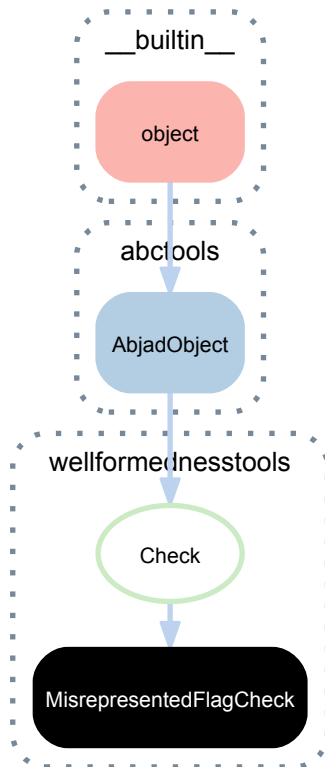
#### Special methods

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject) .**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 43.2.9 wellformednesstools.MisrepresentedFlagCheck



**class** wellformednesstools.**MisrepresentedFlagCheck**

#### Bases

- wellformednesstools.Check
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Methods

(Check) .**check**(*expr*)  
 (Check) .**report**(*expr*)  
 (Check) .**violators**(*expr*)

#### Special methods

(AbjadObject) .**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.

Returns boolean.

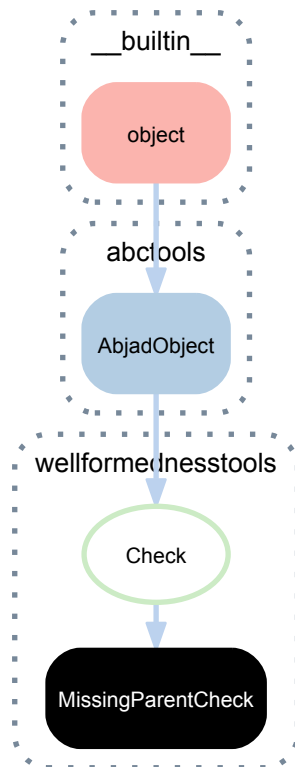
(AbjadObject) .**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject) .**\_\_repr\_\_**()  
Interpreter representation of Abjad object.

Returns string.

### 43.2.10 wellformednesstools.MissingParentCheck



**class** wellformednesstools.**MissingParentCheck**  
Each node except the root needs a parent.

#### Bases

- wellformednesstools.Check
- abctools.AbjadObject
- \_\_builtin\_\_.object

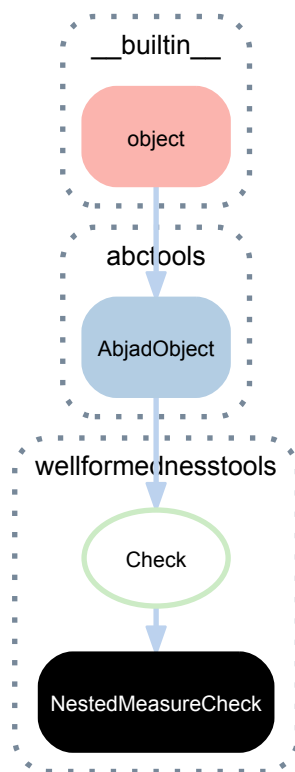
#### Methods

(Check) .**check**(*expr*)  
(Check) .**report**(*expr*)  
(Check) .**violators**(*expr*)

## Special methods

- (AbjadObject) .**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.
- (AbjadObject) .**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.
- (AbjadObject) .**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 43.2.11 wellformednesstools.NestedMeasureCheck



**class** wellformednesstools.NestedMeasureCheck  
 Do we have any nested measures?

## Bases

- wellformednesstools.Check
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Methods

- (Check) .**check**(*expr*)  
 (Check) .**report**(*expr*)

(Check).**violators**(*expr*)

### Special methods

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

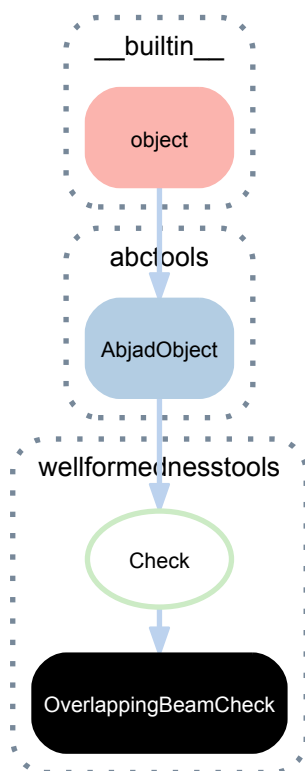
Returns boolean.

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

## 43.2.12 wellformednesstools.OverlappingBeamCheck



**class** wellformednesstools.**OverlappingBeamCheck**  
Beams must not overlap.

### Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`



## Methods

(Check) **.check** (*expr*)

(Check) **.report** (*expr*)

(Check) **.violators** (*expr*)

## Special methods

(AbjadObject) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

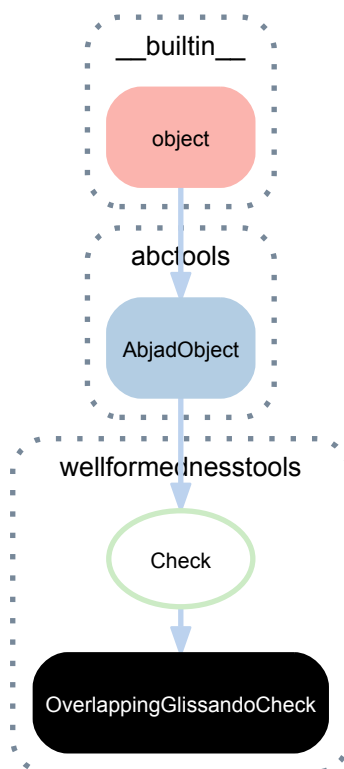
Returns boolean.

(AbjadObject) **.\_\_repr\_\_** ()

Interpreter representation of Abjad object.

Returns string.

### 43.2.13 wellformednesstools.OverlappingGlissandoCheck



**class** `wellformednesstools.OverlappingGlissandoCheck`  
 Glissandi must not overlap. Dove-tailed glissandi are OK.

## Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`

- `__builtin__.object`

## Methods

(Check) **.check** (*expr*)

(Check) **.report** (*expr*)

(Check) **.violators** (*expr*)

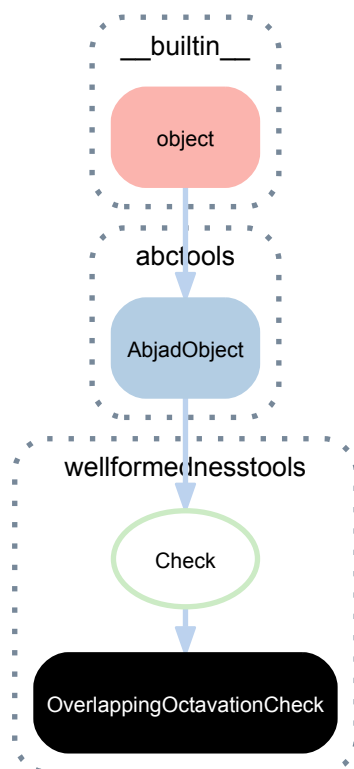
## Special methods

(AbjadObject) **.\_\_eq\_\_** (*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_repr\_\_** ()  
 Interpreter representation of Abjad object.  
 Returns string.

### 43.2.14 wellformednesstools.OverlappingOctavationCheck



**class** `wellformednesstools.OverlappingOctavationCheck`  
 Octavation spanners must not overlap.

## Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

(Check) **.check** (*expr*)

(Check) **.report** (*expr*)

(Check) **.violators** (*expr*)

## Special methods

(AbjadObject) **.\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

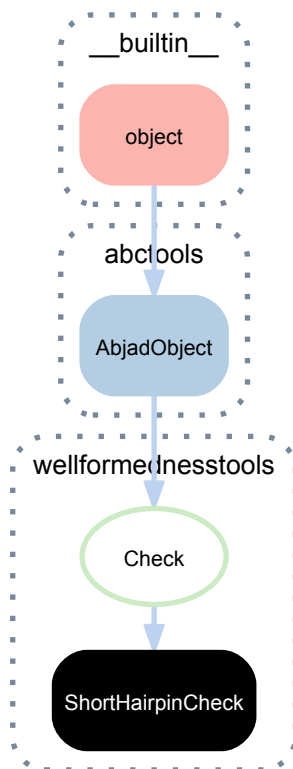
Returns boolean.

(AbjadObject) **.\_\_repr\_\_** ()

Interpreter representation of Abjad object.

Returns string.

### 43.2.15 wellformednesstools.ShortHairpinCheck



**class** `wellformednesstools.ShortHairpinCheck`  
Hairpins must span at least two leaves.

### Bases

- `wellformednesstools.Check`
- `abctools.AbjadObject`
- `__builtin__.object`

### Methods

(`Check`) **.check** (*expr*)

(`Check`) **.report** (*expr*)

(`Check`) **.violators** (*expr*)

### Special methods

(`AbjadObject`) **.\_\_eq\_\_** (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(`AbjadObject`) **.\_\_ne\_\_** (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(`AbjadObject`) **.\_\_repr\_\_** ()  
Interpreter representation of Abjad object.  
Returns string.

## **Part II**

# **Demos and example packages**



# DESORDRE

## 44.1 Functions

### 44.1.1 `desordre.make_desordre_cell`

`desordre.make_desordre_cell` (*pitches*)

The function constructs and returns a *Désordre cell*. *pitches* is a list of numbers or, more generally, pitch tokens.

### 44.1.2 `desordre.make_desordre_lilypond_file`

`desordre.make_desordre_lilypond_file` ()

### 44.1.3 `desordre.make_desordre_measure`

`desordre.make_desordre_measure` (*pitches*)

Constructs a measure composed of *Désordre cells*.

*pitches* is a list of lists of number (e.g., [[1, 2, 3], [2, 3, 4]])

The function returns a measure.

### 44.1.4 `desordre.make_desordre_pitches`

`desordre.make_desordre_pitches` ()

### 44.1.5 `desordre.make_desordre_score`

`desordre.make_desordre_score` (*pitches*)

Returns a complete PianoStaff with Ligeti music!

### 44.1.6 `desordre.make_desordre_staff`

`desordre.make_desordre_staff` (*pitches*)





## FERNEYHOUGH

### 45.1 Functions

#### 45.1.1 `ferneyhough.configure_lilypond_file`

`ferneyhough.configure_lilypond_file` (*lilypond\_file*)

#### 45.1.2 `ferneyhough.configure_score`

`ferneyhough.configure_score` (*score*)

#### 45.1.3 `ferneyhough.make_lilypond_file`

`ferneyhough.make_lilypond_file` (*tuple\_duration*, *row\_count*, *column\_count*)

#### 45.1.4 `ferneyhough.make_nested_tuplet`

`ferneyhough.make_nested_tuplet` (*tuple\_duration*, *outer\_tuplet\_proportions*, *inner\_tuplet\_subdivision\_count*)

#### 45.1.5 `ferneyhough.make_row_of_nested_tuplets`

`ferneyhough.make_row_of_nested_tuplets` (*tuple\_duration*, *outer\_tuplet\_proportions*, *column\_count*)

#### 45.1.6 `ferneyhough.make_rows_of_nested_tuplets`

`ferneyhough.make_rows_of_nested_tuplets` (*tuple\_duration*, *row\_count*, *column\_count*)

#### 45.1.7 `ferneyhough.make_score`

`ferneyhough.make_score` (*tuple\_duration*, *row\_count*, *column\_count*)



# MOZART

## 46.1 Functions

### 46.1.1 `mozart.choose_mozart_measures`

```
mozart.choose_mozart_measures()
```

### 46.1.2 `mozart.make_mozart_lilypond_file`

```
mozart.make_mozart_lilypond_file()
```

### 46.1.3 `mozart.make_mozart_measure`

```
mozart.make_mozart_measure(measure_dict)
```

### 46.1.4 `mozart.make_mozart_measure_corpus`

```
mozart.make_mozart_measure_corpus()
```

### 46.1.5 `mozart.make_mozart_score`

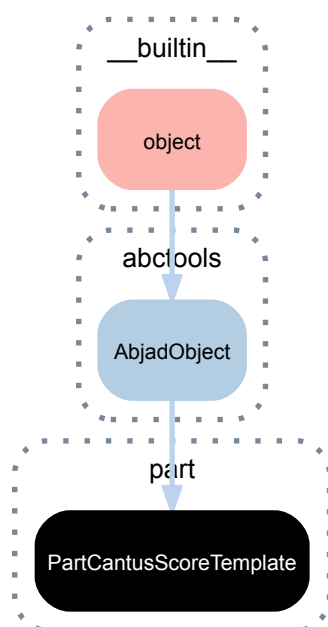
```
mozart.make_mozart_score()
```



## PART

### 47.1 Concrete classes

#### 47.1.1 `part.PartCantusScoreTemplate`



`class part.PartCantusScoreTemplate`

##### Bases

- `abjad.tools.abctools.AbjadObject`
- `__builtin__.object`

##### Special methods

`PartCantusScoreTemplate.__call__()`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject) .**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

## 47.2 Functions

### 47.2.1 `part.add_bell_music_to_score`

`part.add_bell_music_to_score(score)`

### 47.2.2 `part.add_string_music_to_score`

`part.add_string_music_to_score(score)`

### 47.2.3 `part.apply_bowing_marks`

`part.apply_bowing_marks(score)`

### 47.2.4 `part.apply_dynamic_marks`

`part.apply_dynamic_marks(score)`

### 47.2.5 `part.apply_expressive_marks`

`part.apply_expressive_marks(score)`

### 47.2.6 `part.apply_final_bar_lines`

`part.apply_final_bar_lines(score)`

### 47.2.7 `part.apply_page_breaks`

`part.apply_page_breaks(score)`

### 47.2.8 `part.apply_rehearsal_marks`

`part.apply_rehearsal_marks(score)`

### 47.2.9 `part.configure_lilypond_file`

`part.configure_lilypond_file(lilypond_file)`

### 47.2.10 `part.configure_score`

`part.configure_score(score)`

#### 47.2.11 `part.create_pitch_contour_reservoir`

```
part.create_pitch_contour_reservoir()
```

#### 47.2.12 `part.durate_pitch_contour_reservoir`

```
part.durate_pitch_contour_reservoir(pitch_contour_reservoir)
```

#### 47.2.13 `part.edit_bass_voice`

```
part.edit_bass_voice(score, durated_reservoir)
```

#### 47.2.14 `part.edit_cello_voice`

```
part.edit_cello_voice(score, durated_reservoir)
```

#### 47.2.15 `part.edit_first_violin_voice`

```
part.edit_first_violin_voice(score, durated_reservoir)
```

#### 47.2.16 `part.edit_second_violin_voice`

```
part.edit_second_violin_voice(score, durated_reservoir)
```

#### 47.2.17 `part.edit_viola_voice`

```
part.edit_viola_voice(score, durated_reservoir)
```

#### 47.2.18 `part.make_part_lilypond_file`

```
part.make_part_lilypond_file()
```

#### 47.2.19 `part.shadow_pitch_contour_reservoir`

```
part.shadow_pitch_contour_reservoir(pitch_contour_reservoir)
```





## **Part III**

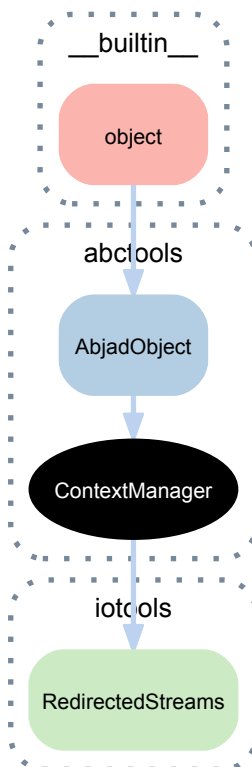
### **Abjad internal packages**



# ABCTOOLS

## 48.1 Abstract classes

### 48.1.1 abctools.ContextManager



**class** `abctools.ContextManager`  
An abstract context manager class.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`ContextManager.__enter__()`

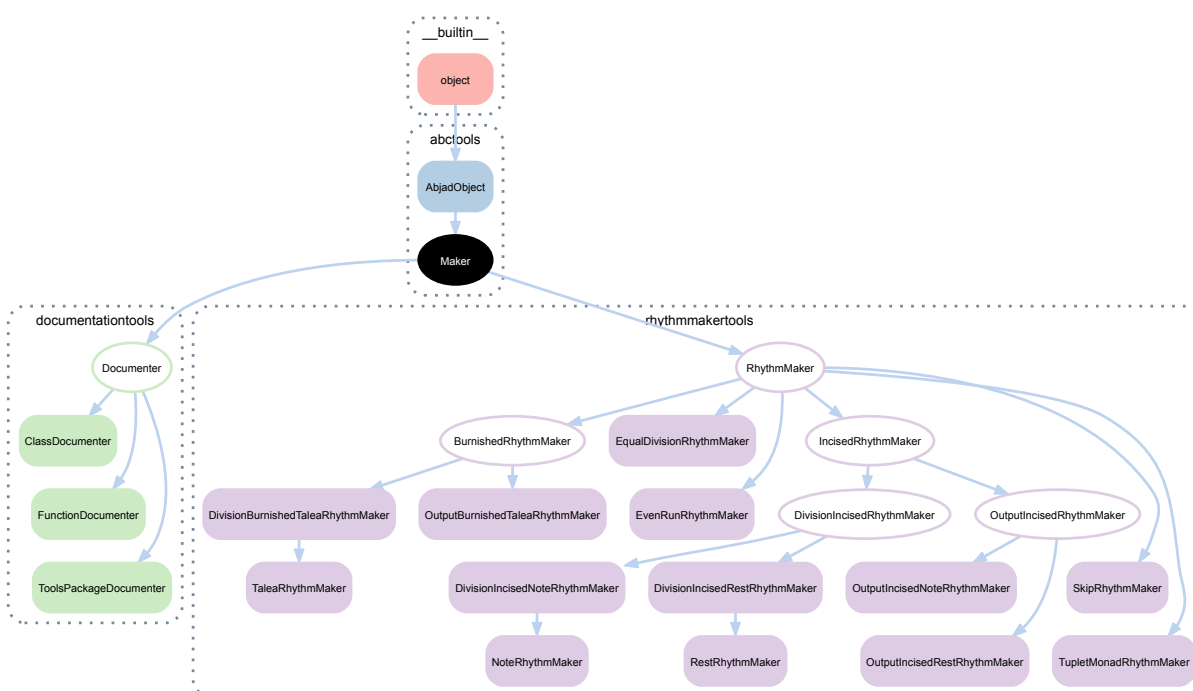
(AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

ContextManager.**\_\_exit\_\_**()

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

## 48.1.2 abctools.Maker



**class abctools.Maker**  
 Abstract base class for all maker classes.

### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

**Maker.storage\_format**  
 Storage format of maker.  
 Returns string.

## Methods

`Maker.new()`

## Special methods

`Maker.__call__()`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

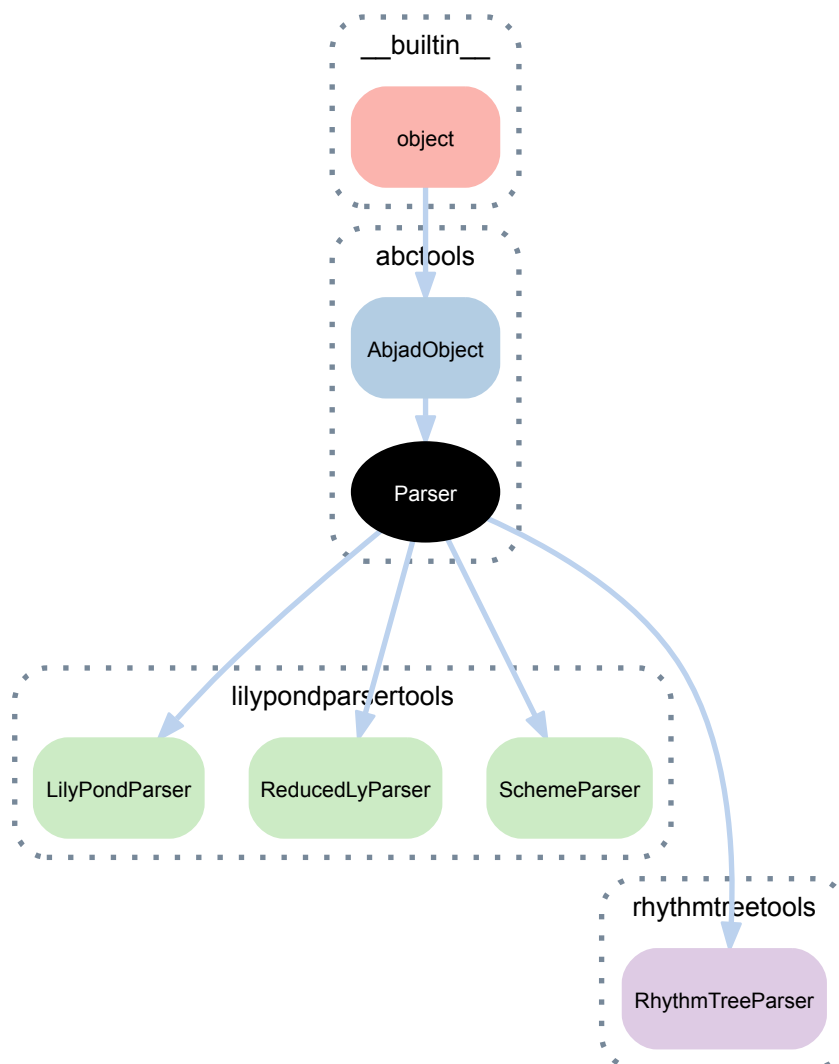
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 48.1.3 abctools.Parser



**class** `abctools.Parser` (*debug=False*)

Abstract base class for Abjad parsers.

Rules objects for lexing and parsing must be defined by overriding the abstract properties *lexer\_rules\_object* and *parser\_rules\_object*.

For most parsers these properties should simply return *self*.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`Parser.debug`

True if the parser runs in debugging mode.

`Parser.lexer`

The parser's PLY Lexer instance.

`Parser.lexer_rules_object`

The object containing the parser's lexical rule definitions.

`Parser.logger`

The parser's Logger instance.

`Parser.logger_path`

The output path for the parser's logfile.

`Parser.output_path`

The output path for files associated with the parser.

`Parser.parser`

The parser's PLY LRParser instance.

`Parser.parser_rules_object`

The object containing the parser's syntactical rule definitions.

`Parser.pickle_path`

The output path for the parser's pickled parsing tables.

## Methods

`Parser.tokenize` (*input\_string*)

Tokenize *input\_string* and print results.

## Special methods

`Parser.__call__` (*input\_string*)

Parse *input\_string* and return result.

(`AbjadObject`) `.__eq__` (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(`AbjadObject`) `.__ne__` (*expr*)

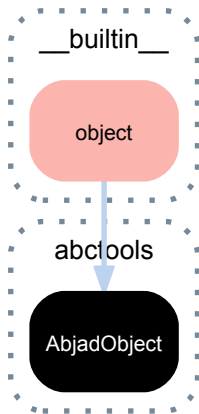
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

## 48.2 Concrete classes

### 48.2.1 abctools.AbjadObject



**class** `abctools.AbjadObject`  
 Abstract base class from which all custom classes should inherit.  
 Abjad objects compare equal only with equal object IDs.

#### Bases

- `__builtin__.object`

#### Special methods

`AbjadObject.__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`AbjadObject.__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`AbjadObject.__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

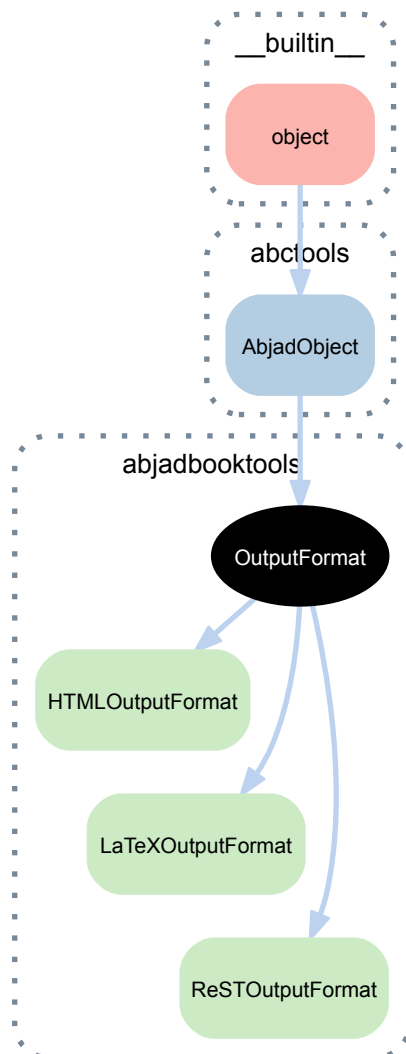




# ABJADBOOKTOOLS

## 49.1 Abstract classes

### 49.1.1 abjadbooktools.OutputFormat



```
class abjadbooktools.OutputFormat (code_block_opening, code_block_closing, code_indent,  
                                     image_block, image_format)
```

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`OutputFormat.code_block_closing`

`OutputFormat.code_block_opening`

`OutputFormat.code_indent`

`OutputFormat.image_block`

`OutputFormat.image_format`

## Special methods

`OutputFormat.__call__(code_block, image_dict)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

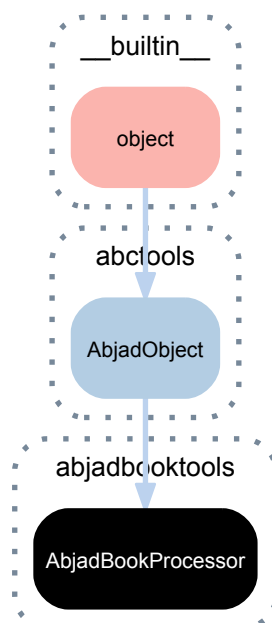
`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 49.2 Concrete classes

### 49.2.1 abjadbooktools.AbjadBookProcessor



```
class abjadbooktools.AbjadBookProcessor(directory, lines, output_format,
                                         skip_rendering=False, image_prefix='image',
                                         verbose=False)
```

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`AbjadBookProcessor.directory`

`AbjadBookProcessor.image_prefix`

`AbjadBookProcessor.lines`

`AbjadBookProcessor.output_format`

`AbjadBookProcessor.skip_rendering`

`AbjadBookProcessor.verbose`

## Methods

`AbjadBookProcessor.update_status(line)`

## Special methods

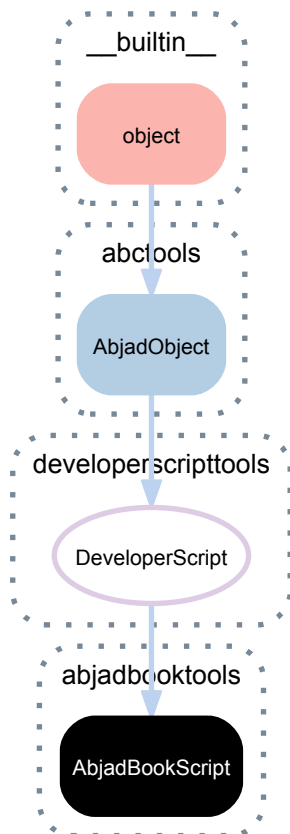
`AbjadBookProcessor.__call__(verbose=True)`

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

### 49.2.2 abjadbooktools.AbjadBookScript



**class** `abjadbooktools.AbjadBookScript`

#### Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`AbjadBookScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`AbjadBookScript.long_description`

`AbjadBookScript.output_formats`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`(DeveloperScript).scripting_group`

The script's scripting subcommand group.

AbjadBookScript.**short\_description**

AbjadBookScript.**version**

## Methods

AbjadBookScript.**process\_args** (*args*)

AbjadBookScript.**setup\_argument\_parser** (*parser*)

## Special methods

(DeveloperScript).**\_\_call\_\_** (*args=None*)

(AbjadObject).**\_\_eq\_\_** (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_** (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

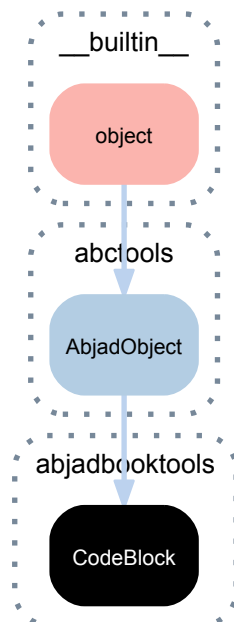
Returns boolean.

(AbjadObject).**\_\_repr\_\_** ()

Interpreter representation of Abjad object.

Returns string.

## 49.2.3 abjadbooktools.CodeBlock



**class** abjadbooktools.**CodeBlock** (*lines, starting\_line\_number, ending\_line\_number, hide=False, strip\_prompt=False*)

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`CodeBlock.ending_line_number`  
`CodeBlock.hide`  
`CodeBlock.lines`  
`CodeBlock.processed_results`  
`CodeBlock.starting_line_number`  
`CodeBlock.strip_prompt`

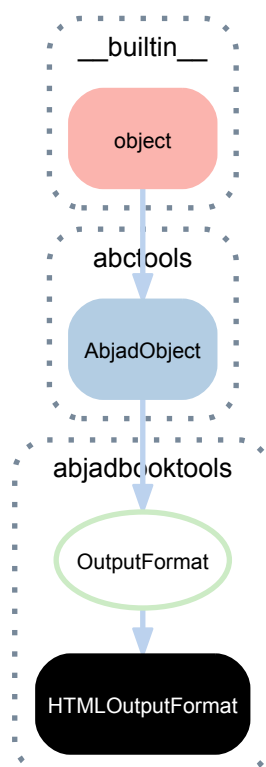
## Methods

`CodeBlock.read(pipe)`

## Special methods

`CodeBlock.__call__` (*processor, pipe, image\_count=0, directory=None, image\_prefix='image', verbose=False*)  
`CodeBlock.__eq__` (*expr*)  
 (`AbjadObject`) `.__ne__` (*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.  
 (`AbjadObject`) `.__repr__` ()  
 Interpreter representation of Abjad object.  
 Returns string.

### 49.2.4 abjadbooktools.HTMLOutputFormat



**class** abjadbooktools.**HTMLOutputFormat**

### Bases

- abjadbooktools.OutputFormat
- abctools.AbjadObject
- `__builtin__.object`

### Read-only properties

(OutputFormat).**code\_block\_closing**

(OutputFormat).**code\_block\_opening**

(OutputFormat).**code\_indent**

(OutputFormat).**image\_block**

(OutputFormat).**image\_format**

### Special methods

(OutputFormat).**\_\_call\_\_**(*code\_block*, *image\_dict*)

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

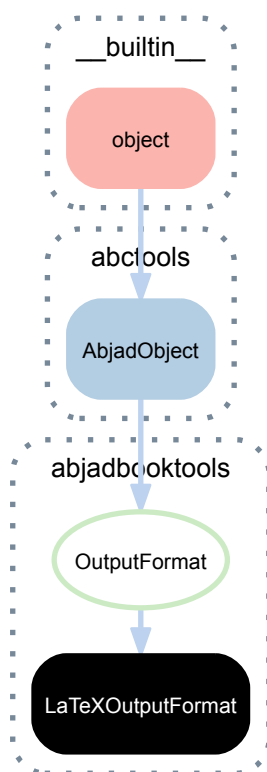
Returns boolean.

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

### 49.2.5 abjadbooktools.LaTeXOutputFormat



**class** `abjadbooktools.LaTeXOutputFormat`

#### Bases

- `abjadbooktools.OutputFormat`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(OutputFormat).code_block_closing`

`(OutputFormat).code_block_opening`

`(OutputFormat).code_indent`

`(OutputFormat).image_block`

`(OutputFormat).image_format`

#### Special methods

`(OutputFormat).__call__(code_block, image_dict)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.



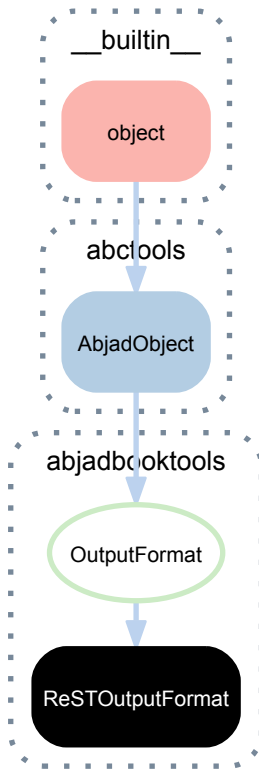
Returns boolean.

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

## 49.2.6 abjadbooktools.ReSTOutputFormat



**class** abjadbooktools.ReSTOutputFormat

### Bases

- abjadbooktools.OutputFormat
- abctools.AbjadObject
- \_\_builtin\_\_.object

### Read-only properties

(OutputFormat).**code\_block\_closing**

(OutputFormat).**code\_block\_opening**

(OutputFormat).**code\_indent**

(OutputFormat).**image\_block**

(OutputFormat).**image\_format**

### Special methods

(OutputFormat).**\_\_call\_\_**(code\_block, image\_dict)

(AbjadObject) .**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject) .**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject) .**\_\_repr\_\_**()

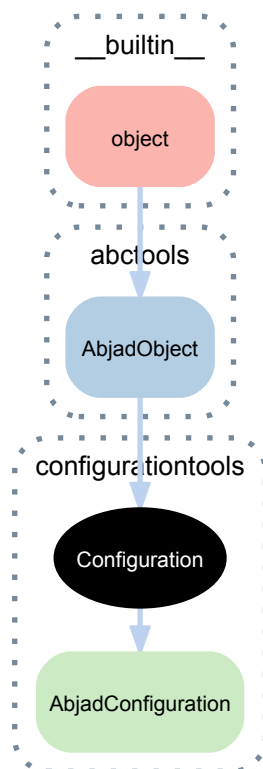
Interpreter representation of Abjad object.

Returns string.

# CONFIGURATIONTOOLS

## 50.1 Abstract classes

### 50.1.1 configurationtools.Configuration



**class** `configurationtools.Configuration`  
A configuration object.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`Configuration.configuration_directory_path`  
Configuration directory path.

Returns string.

`Configuration.configuration_file_name`  
Configuration file name.

Returns string.

`Configuration.configuration_file_path`  
Configuration file path.

Returns string.

`Configuration.home_directory_path`  
Home directory path.

Returns string.

## Special methods

`Configuration.__delitem__(i)`  
Deletes *i* from settings.

Returns none.

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`Configuration.__getitem__(i)`  
Gets *i* from settings.

Returns none.

`Configuration.__iter__()`  
Iterates settings.

Returns generator.

`Configuration.__len__()`  
Number of settings in configuration.

Returns nonnegative integer.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.

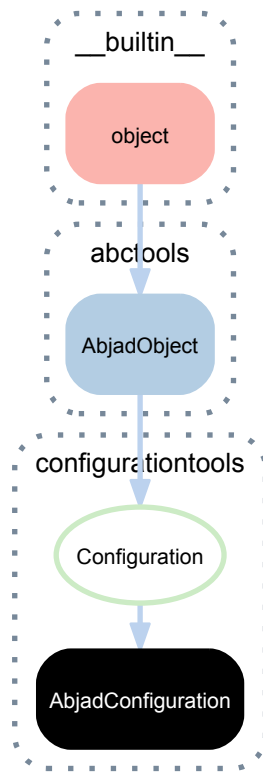
Returns string.

`Configuration.__setitem__(i, arg)`  
Sets setting *i* to *arg*.

Returns none.

## 50.2 Concrete classes

### 50.2.1 configurationtools.AbjadConfiguration



**class** configurationtools.**AbjadConfiguration**  
 Abjad configuration.

```
>>> ABJCONFIG = configurationtools.AbjadConfiguration()
>>> ABJCONFIG['accidental_spelling']
'mixed'
```

```
>>> configuration = configurationtools.AbjadConfiguration()
```

*AbjadConfiguration* creates the *\$HOME/.abjad/* directory on instantiation.

*AbjadConfiguration* then attempts to read an *abjad.cfg* file in that directory and parse the file as a *ConfigObj* configuration. *AbjadConfiguration* generates a default configuration if no file is found.

*AbjadConfiguration* validates the *ConfigObj* instance and replaces key-value pairs which fail validation with default values. *AbjadConfiguration* then writes the configuration back to disk.

The Abjad output directory is created the from *abjad\_output* key if it does not already exist.

*AbjadConfiguration* supports the mutable mapping interface and can be subscripted as a dictionary.

#### Bases

- configurationtools.Configuration
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

`AbjadConfiguration.abjad_configuration_directory_path`  
Abjad configuration directory path.

Returns string.

`AbjadConfiguration.abjad_configuration_file_path`  
Abjad configuration file path.

Returns string.

`AbjadConfiguration.abjad_directory_path`  
Abjad directory path.

Returns string.

`AbjadConfiguration.abjad_experimental_directory_path`  
Abjad experimental directory path.

Returns string.

`AbjadConfiguration.abjad_output_directory_path`  
Abjad output directory path.

Returns string.

`AbjadConfiguration.abjad_root_directory_path`  
Abjad root directory path.

Returns string.

`AbjadConfiguration.configuration_directory_path`  
Configuration directory path.

Returns string.

`AbjadConfiguration.configuration_file_name`  
Configuration file name.

Returns string.

`(Configuration).configuration_file_path`  
Configuration file path.

Returns string.

`(Configuration).home_directory_path`  
Home directory path.

Returns string.

## Class methods

`AbjadConfiguration.get_abjad_startup_string()`  
Gets Abjad startup string.

```
>>> abjad_configuration.get_abjad_startup_string()
'Abjad 2.13 (r12069)'
```

Returns string.

`AbjadConfiguration.get_lilypond_minimum_version_string()`  
Gets LilyPond minimum version string.

```
>>> abjad_configuration.get_lilypond_minimum_version_string()
'2.17.0'
```

This is useful for documentation purposes, where all developers are using the development version of LilyPond, but not necessarily the exact same version.

Returns string.

## Static methods

`AbjadConfiguration.get_abjad_revision_string()`  
Gets Abjad revision string.

```
>>> abjad_configuration.get_abjad_revision_string()
'11266'
```

Returns string.

`AbjadConfiguration.get_abjad_version_string()`  
Gets Abjad version string.

```
>>> abjad_configuration.get_abjad_version_string()
'2.13'
```

Returns string.

`AbjadConfiguration.get_lilypond_version_string()`  
Gets LilyPond version string:

```
>>> abjad_configuration.get_lilypond_version_string()
'2.13.61'
```

Returns string.

`AbjadConfiguration.get_python_version_string()`  
Gets Python version string.

```
>>> abjad_configuration.get_python_version_string()
'2.6.1'
```

Returns string.

`AbjadConfiguration.get_tab_width()`  
Gets tab width.

```
>>> abjad_configuration.get_tab_width()
4
```

The value is used by various functions that generate or test code in the system.

Returns nonnegative integer.

`AbjadConfiguration.get_text_editor()`  
Get text editor.

```
>>> abjad_configuration.get_text_editor()
'vim'
```

Returns string.

`AbjadConfiguration.list_abjad_environment_variables()`  
Lists Abjad environment variables.

```
>>> for x in abjad_configuration.list_abjad_environment_variables():
...     x
```

Abjad environment variables are defined in `abjad/tools/abjad_configuration/AbjadConfiguration.py`.

Returns tuple of zero or more environment variable / setting pairs.

`AbjadConfiguration.list_package_dependency_versions()`  
Lists package dependency versions.

```
>>> abjad_configuration.list_package_dependency_versions()
{'sphinx': '1.1.2', 'py.test': '2.1.2'}
```

Returns dictionary.

`AbjadConfiguration.set_default_accidental_spelling(spelling='mixed')`  
 Set default accidental spelling to sharps:

```
>>> abjad_configuration.set_default_accidental_spelling('sharps')
```

```
>>> [Note(13, (1, 4)), Note(15, (1, 4))]
[Note("cs''4"), Note("ds''4")]
```

Set default accidental spelling to flats:

```
>>> abjad_configuration.set_default_accidental_spelling('flats')
```

```
>>> [Note(13, (1, 4)), Note(15, (1, 4))]
[Note("df''4"), Note("ef''4")]
```

Set default accidental spelling to mixed:

```
>>> abjad_configuration.set_default_accidental_spelling()
```

```
>>> [Note(13, (1, 4)), Note(15, (1, 4))]
[Note("cs''4"), Note("ef''4")]
```

Mixed is system default.

Mixed test case must appear last here for doc tests to check correctly.

Returns none.

## Special methods

`(Configuration).__delitem__(i)`  
 Deletes *i* from settings.

Returns none.

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(Configuration).__getitem__(i)`  
 Gets *i* from settings.

Returns none.

`(Configuration).__iter__()`  
 Iterates settings.

Returns generator.

`(Configuration).__len__()`  
 Number of settings in configuration.

Returns nonnegative integer.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.

Returns string.



(Configuration).\_\_setitem\_\_(*i*, *arg*)

Sets setting *i* to *arg*.

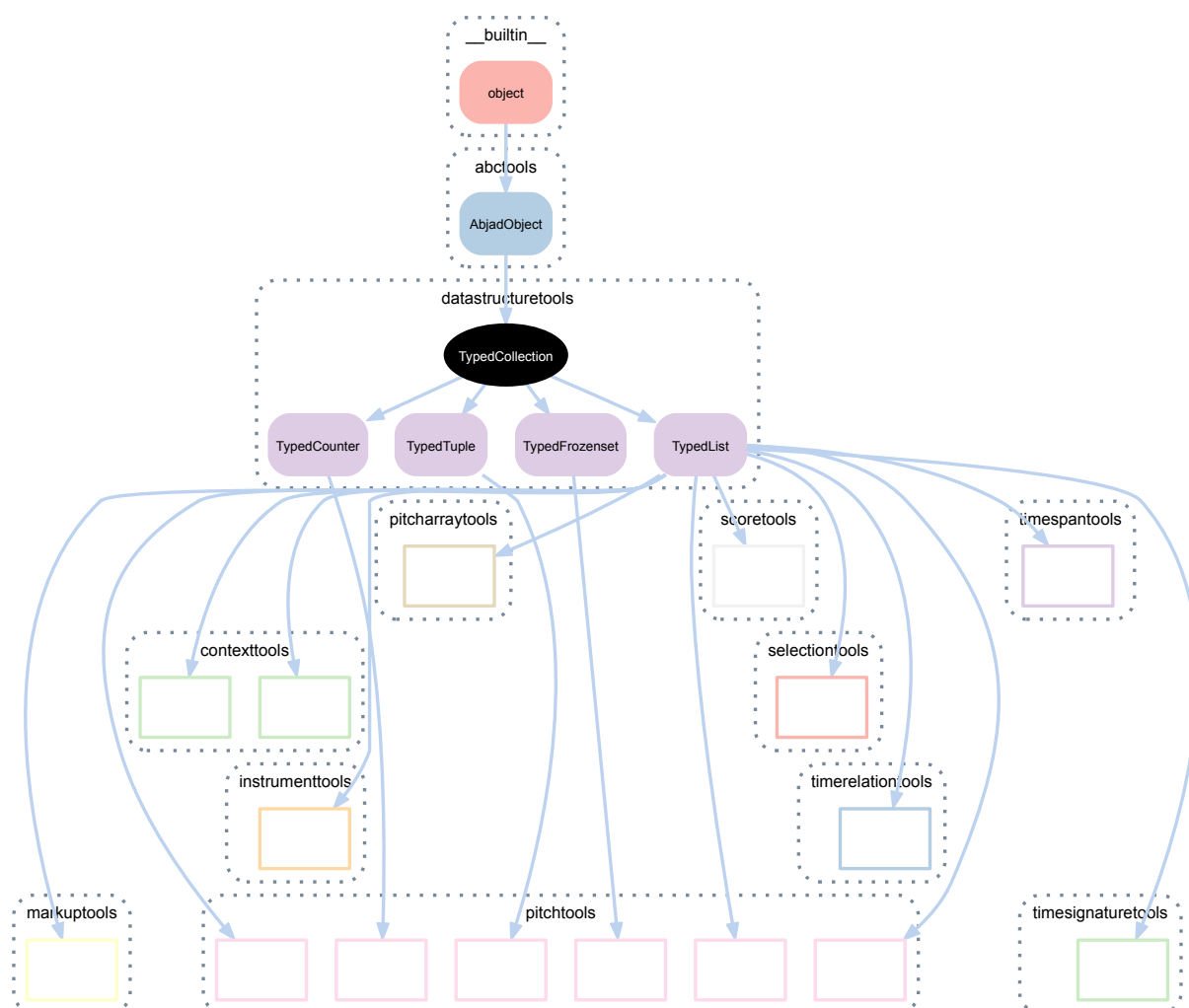
Returns none.



# DATASTRUCTURETOOLS

## 51.1 Abstract classes

### 51.1.1 `datastructuretools.TypedCollection`



**class** `datastructuretools.TypedCollection` (*tokens=None, item\_class=None, name=None*)

#### Bases

- `abctools.AbjadObject`

- `__builtin__.object`

### Read-only properties

`TypedCollection.item_class`

Item class to coerce tokens into.

`TypedCollection.storage_format`

Storage format of typed tuple.

### Read/write properties

`TypedCollection.name`

Read / write name of typed tuple.

### Methods

`TypedCollection.new` (*tokens=None, item\_class=None, name=None*)

### Special methods

`TypedCollection.__contains__` (*token*)

`TypedCollection.__eq__` (*expr*)

`TypedCollection.__iter__` ()

`TypedCollection.__len__` ()

`TypedCollection.__ne__` (*expr*)

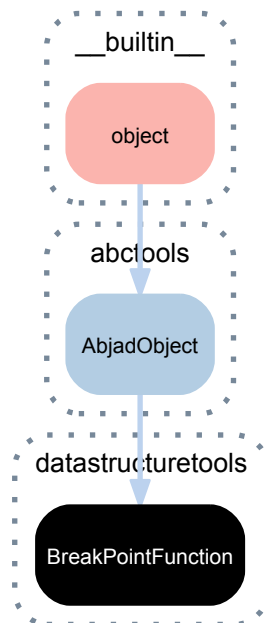
`(AbjadObject).__repr__` ()

Interpreter representation of Abjad object.

Returns string.

## 51.2 Concrete classes

### 51.2.1 datastructuretools.BreakPointFunction



**class** datastructuretools.**BreakPointFunction** (*bpf*)  
 A break-point function:

```
>>> bpf = datastructuretools.BreakPointFunction({
...     0.: 0.,
...     0.75: (-1, 1.),
...     1.: 0.25,
...     })
```

Allows interpolated lookup, and supports discontinuities on the y-axis:

```
>>> for x in (-0.5, 0., 0.25, 0.5, 0.7499, 0.75, 1., 1.5):
...     bpf[x]
0.0
0.0
-0.3333333333333333
-0.6666666666666666
-0.9998666666666667
1.0
0.25
0.25
```

Returns break-point function instance.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`BreakPointFunction.bpf`

A copy of the `BreakPointFunction`'s internal data-structure:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).bpf  
{0.0: (0.25,), 0.5: (1.3,), 1.0: (0.9,)}
```

Returns dict.

**BreakPointFunction.dc\_bias**

The mean y-value of a BreakPointFunction:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0., 0.25: 0., 0.5: (0.75, 0.25), 1.: 1.}  
...     ).dc_bias  
0.4
```

Returns number.

**BreakPointFunction.gnuplot\_format**

**BreakPointFunction.x\_center**

The arithmetic mean of a BreakPointFunction's x-range:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).x_center  
0.5
```

Returns number.

**BreakPointFunction.x\_range**

The minimum and maximum x-values of a BreakPointFunction:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).x_range  
(0.0, 1.0)
```

Returns pair.

**BreakPointFunction.x\_values**

The sorted x-values of a BreakPointFunction:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).x_values  
(0.0, 0.5, 1.0)
```

Returns tuple.

**BreakPointFunction.y\_center**

The arithmetic mean of a BreakPointFunction's y-range:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).y_center  
0.775
```

Returns number.

**BreakPointFunction.y\_range**

The minimum and maximum y-values of a BreakPointFunction:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).y_range  
(0.25, 1.3)
```

Returns pair.

## Methods

**BreakPointFunction.clip\_x\_axis** (*minimum=0, maximum=1*)

Clip x-axis between *minimum* and *maximum*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 1., 1.: 0.})
>>> bpf.clip_x_axis(minimum=0.25, maximum=0.75)
BreakPointFunction({
    0.25: (0.75,),
    0.75: (0.25,)
})
```

Emit new *BreakPointFunction* instance.

*BreakPointFunction*.**clip\_y\_axis** (*minimum=0, maximum=1*)  
Clip y-axis between *minimum* and *maximum*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 1., 1.: 0.})
>>> bpf.clip_y_axis(minimum=0.25, maximum=0.75)
BreakPointFunction({
    0.0: (0.75,),
    1.0: (0.25,)
})
```

Emit new *BreakPointFunction* instance.

*BreakPointFunction*.**concatenate** (*expr*)  
Concatenate self with *expr*:

```
>>> one = datastructuretools.BreakPointFunction(
...     {0.0: 0.0, 1.0: 1.0})
>>> two = datastructuretools.BreakPointFunction(
...     {0.5: 0.75, 1.5: 0.25})
```

```
>>> one.concatenate(two)
BreakPointFunction({
    0.0: (0.0,),
    1.0: (1.0, 0.75),
    2.0: (0.25,)
})
```

Emit new *BreakPointFunction* instance.

*BreakPointFunction*.**get\_y\_at\_x** (*x*)  
Get y-value at *x*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.5: (-1., 1.), 1.: 0.5})
```

```
>>> bpf.get_y_at_x(-1000)
0.0
```

```
>>> bpf.get_y_at_x(0.)
0.0
```

```
>>> bpf.get_y_at_x(0.25)
-0.5
```

```
>>> bpf.get_y_at_x(0.4999)
-0.9998
```

```
>>> bpf.get_y_at_x(0.5)
1.0
```

```
>>> bpf.get_y_at_x(0.75)
0.75
```

```
>>> bpf.get_y_at_x(1.)
0.5
```

```
>>> bpf.get_y_at_x(1000)
0.5
```

Returns Number.

`BreakPointFunction.invert` (*y\_center=None*)

Invert self:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0., 1.: 1.}).invert()  
BreakPointFunction(  
    0.0: (1.0,),  
    1.0: (0.0,) )
```

If *y\_center* is not None, use *y\_center* as the axis of inversion:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0., 1.: 1.}).invert(0)  
BreakPointFunction(  
    0.0: (0.0,),  
    1.0: (-1.0,) )
```

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0., 1.: 1.}).invert(0.25)  
BreakPointFunction(  
    0.0: (0.5,),  
    1.0: (-0.5,) )
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.normalize_axes` ()

Scale both x and y axes between 0 and 1:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.25: 0.25, 0.75: 0.75}).normalize_axes()  
BreakPointFunction(  
    0.0: (0.0,),  
    1.0: (1.0,) )
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.reflect` (*x\_center=None*)

Reflect x values of a *BreakPointFunction*:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.25: 0., 0.5: (-1., 2.), 1: 1.}).reflect()  
BreakPointFunction(  
    0.25: (1.0,),  
    0.75: (2.0, -1.0),  
    1.0: (0.0,) )
```

If *x\_center* is not None, reflection will take *x\_center* as the axis of reflection:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.25: 0., 0.5: (-1., 2.), 1: 1.}).reflect(x_center=0.25)  
BreakPointFunction(  
    -0.5: (1.0,),  
    0.0: (2.0, -1.0),  
    0.25: (0.0,) )
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.remove_dc_bias` ()

Remove dc-bias from a *BreakPointFunction*:

```
>>> datastructuretools.BreakPointFunction(  
...     {0.: 0., 1.: 1.}).remove_dc_bias()  
BreakPointFunction(  
    0.0: (0.0,),  
    1.0: (1.0,) )
```



```

    0.0: (-0.5,),
    1.0: (0.5,)
})

```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.scale_x_axis` (*minimum=0, maximum=1*)

Scale x-axis between *minimum* and *maximum*:

```

>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 0.5: (-1., 2.), 1.: 1.}
...     ).scale_x_axis(-2, 2)
BreakPointFunction({
    -2.0: (0.0,),
    0.0: (-1.0, 2.0),
    2.0: (1.0,)
})

```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.scale_y_axis` (*minimum=0, maximum=1*)

Scale y-axis between *minimum* and *maximum*:

```

>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 0.5: (-1., 2.), 1.: 1.}
...     ).scale_y_axis(-2, 4)
BreakPointFunction({
    0.0: (0.0,),
    0.5: (-2.0, 4.0),
    1.0: (2.0,)
})

```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.set_y_at_x` (*x, y*)

Set y-value at *x*:

```

>>> bpf = datastructuretools.BreakPointFunction(
...     {0.0: 0.0, 1.0: 1.0})

```

With a number:

```

>>> bpf.set_y_at_x(0.25, 0.75)
>>> bpf
BreakPointFunction({
    0.0: (0.0,),
    0.25: (0.75,),
    1.0: (1.0,)
})

```

With a pair:

```

>>> bpf.set_y_at_x(0.6, (-2., 2.))
>>> bpf
BreakPointFunction({
    0.0: (0.0,),
    0.25: (0.75,),
    0.6: (-2.0, 2.0),
    1.0: (1.0,)
})

```

Delete all values at *x* when *y* is `None`:

```

>>> bpf.set_y_at_x(0., None)
>>> bpf
BreakPointFunction({
    0.25: (0.75,),
    0.6: (-2.0, 2.0),
    1.0: (1.0,)
})

```

Operates in place and returns None.

`BreakPointFunction.tessalate_by_ratio` (*ratio*, *invert\_on\_negative=False*, *reflect\_on\_negative=False*, *y\_center=None*) *re-*

Concatenate copies of a `BreakPointFunction`, stretched by the weights in *ratio*:

```
>>> bpf = datastructuretools.BreakPointFunction(  
...     {0.: 0., 0.25: 0.9, 1.: 1.})
```

```
>>> bpf.tessalate_by_ratio((1, 2, 3))  
BreakPointFunction(  
    0.0: (0.0,),  
    0.25: (0.9,),  
    1.0: (1.0, 0.0),  
    1.5: (0.9,),  
    3.0: (1.0, 0.0),  
    3.75: (0.9,),  
    6.0: (1.0,) )
```

Negative ratio values are still treated as weights.

If *invert\_on\_negative* is `True`, copies corresponding to negative ratio values will be inverted:

```
>>> bpf.tessalate_by_ratio((1, -2, 3), invert_on_negative=True)  
BreakPointFunction(  
    0.0: (0.0,),  
    0.25: (0.9,),  
    1.0: (1.0,),  
    1.5: (0.09999999999999998,),  
    3.0: (0.0,),  
    3.75: (0.9,),  
    6.0: (1.0,) )
```

If *y\_center* is not `None`, inversion will take *y\_center* as the axis of inversion:

```
>>> bpf.tessalate_by_ratio((1, -2, 3),  
...     invert_on_negative=True, y_center=0)  
BreakPointFunction(  
    0.0: (0.0,),  
    0.25: (0.9,),  
    1.0: (1.0, 0.0),  
    1.5: (-0.9,),  
    3.0: (-1.0, 0.0),  
    3.75: (0.9,),  
    6.0: (1.0,) )
```

If *reflect\_on\_negative* is `True`, copies corresponding to negative ratio values will be reflected:

```
>>> bpf.tessalate_by_ratio((1, -2, 3), reflect_on_negative=True)  
BreakPointFunction(  
    0.0: (0.0,),  
    0.25: (0.9,),  
    1.0: (1.0,),  
    2.5: (0.9,),  
    3.0: (0.0,),  
    3.75: (0.9,),  
    6.0: (1.0,) )
```

Inversion may be combined reflecting.

Emit new `BreakPointFunction` instance.

## Special methods

`BreakPointFunction.__add__` (*expr*)

Add *expr* to all y-values in self:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.75: (-1., 1.), 1.: 0.25})
>>> bpf + 0.3
BreakPointFunction({
    0.0: (0.3,),
    0.75: (-0.7, 1.3),
    1.0: (0.55,)
})
```

*expr* may also be a *BreakPointFunction* instance:

```
>>> bpf2 = datastructuretools.BreakPointFunction({0.: 1., 1.: 0.})
>>> bpf + bpf2
BreakPointFunction({
    0.0: (1.0,),
    0.75: (-0.75, 1.25),
    1.0: (0.25,)
})
```

Emit new *BreakPointFunction*.

*BreakPointFunction*.**\_\_div\_\_**(*expr*)

Divide y-values in self by *expr*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.75: (-1., 1.), 1.: 0.25})
>>> bpf / 2.
BreakPointFunction({
    0.0: (0.0,),
    0.75: (-0.5, 0.5),
    1.0: (0.125,)
})
```

*expr* may also be a *BreakPointFunction* instance.

Emit new *BreakPointFunction*.

(*AbjadObject*).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

*BreakPointFunction*.**\_\_getitem\_\_**(*item*)

Aliases *BreakPointFunction*.get\_y\_at\_x().

*BreakPointFunction*.**\_\_mul\_\_**(*expr*)

Multiply y-values in self by *expr*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.75: (-1., 1.), 1.: 0.25})
>>> bpf * 2.
BreakPointFunction({
    0.0: (0.0,),
    0.75: (-2.0, 2.0),
    1.0: (0.5,)
})
```

*expr* may also be a *BreakPointFunction* instance.

Emit new *BreakPointFunction*.

(*AbjadObject*).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

*BreakPointFunction*.**\_\_repr\_\_**()

*BreakPointFunction*.**\_\_sub\_\_**(*expr*)

Subtract *expr* from all y-values in self:

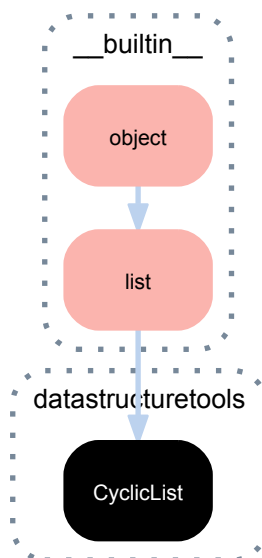
```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.75: (-1., 1.), 1.: 0.25})
>>> bpf - 0.3
BreakPointFunction({
    0.0: (-0.3,),
    0.75: (-1.3, 0.7),
    1.0: (-0.04999999999999999,)
})
```

*expr* may also be a *BreakPointFunction* instance:

```
>>> bpf2 = datastructuretools.BreakPointFunction(
...     {0.: 1., 1.: 0.})
>>> bpf - bpf2
BreakPointFunction({
    0.0: (-1.0,),
    0.75: (-1.25, 0.75),
    1.0: (0.25,)
})
```

Emit new *BreakPointFunction*.

## 51.2.2 datastructuretools.CyclicList



**class datastructuretools.CyclicList**

Abjad model of cyclic list:

```
>>> cyclic_list = datastructuretools.CyclicList('abcd')
```

```
>>> cyclic_list
CyclicList([a, b, c, d])
```

```
>>> for x in range(8):
...     print x, cyclic_list[x]
...
0 a
1 b
2 c
3 d
4 a
5 b
6 c
7 d
```

Cyclic lists overload the item-getting method of built-in lists.

Cyclic lists return a value for any integer index.

Cyclic lists otherwise behave exactly like built-in lists.

## Bases

- `__builtin__.list`
- `__builtin__.object`

## Methods

```
(list) .append()
    L.append(object) – append object to end

(list) .count (value) → integer – return number of occurrences of value

(list) .extend()
    L.extend(iterable) – extend list by appending elements from the iterable

(list) .index (value[, start[, stop]]) → integer – return first index of value.
    Raises ValueError if the value is not present.

(list) .insert ()
    L.insert(index, object) – insert object before index

(list) .pop ([index]) → item – remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.

(list) .remove ()
    L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

(list) .reverse ()
    L.reverse() – reverse IN PLACE

(list) .sort ()
    L.sort(cmp=None, key=None, reverse=False) – stable sort IN PLACE; cmp(x, y) -> -1, 0, 1
```

## Special methods

```
(list) .__add__ ()
    x.__add__(y) <==> x+y

(list) .__contains__ ()
    x.__contains__(y) <==> y in x

(list) .__delitem__ ()
    x.__delitem__(y) <==> del x[y]

(list) .__delslice__ ()
    x.__delslice__(i, j) <==> del x[i:j]
    Use of negative indices is not supported.

(list) .__eq__ ()
    x.__eq__(y) <==> x==y

(list) .__ge__ ()
    x.__ge__(y) <==> x>=y

CyclicList .__getitem__ (expr)

CyclicList .__getslice__ (start_index, stop_index)

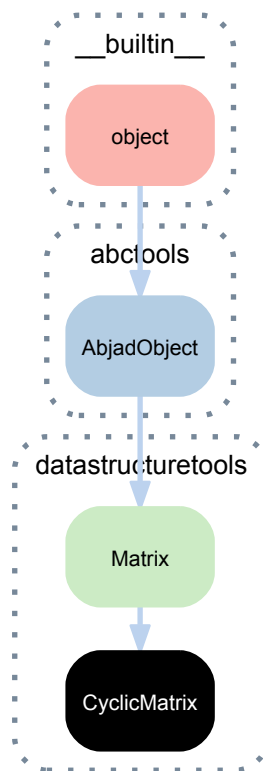
(list) .__gt__ ()
    x.__gt__(y) <==> x>y
```

```

(list) .__iadd__()
    x.__iadd__(y) <==> x+=y
(list) .__imul__()
    x.__imul__(y) <==> x*=y
(list) .__iter__() <==> iter(x)
(list) .__le__()
    x.__le__(y) <==> x<=y
(list) .__len__() <==> len(x)
(list) .__lt__()
    x.__lt__(y) <==> x<y
(list) .__mul__()
    x.__mul__(n) <==> x*n
(list) .__ne__()
    x.__ne__(y) <==> x!=y
CyclicList.__repr__()
(list) .__reversed__()
    L.__reversed__() – return a reverse iterator over the list
(list) .__rmul__()
    x.__rmul__(n) <==> n*x
(list) .__setitem__()
    x.__setitem__(i, y) <==> x[i]=y
(list) .__setslice__()
    x.__setslice__(i, j, y) <==> x[i:j]=y
    Use of negative indices is not supported.
CyclicList.__str__()

```

### 51.2.3 datastructuretools.CyclicMatrix



**class** datastructuretools.**CyclicMatrix**(\*args, \*\*kwargs)  
 Abjad model of cyclic matrix.

Initialize from rows:

```
>>> cyclic_matrix = datastructuretools.CyclicMatrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> cyclic_matrix
CyclicMatrix(3x4)
```

```
>>> cyclic_matrix[2]
CyclicTuple([20, 21, 22, 23])
```

```
>>> cyclic_matrix[2][2]
22
```

```
>>> cyclic_matrix[99]
CyclicTuple([0, 1, 2, 3])
```

```
>>> cyclic_matrix[99][99]
3
```

Initialize from columns:

```
>>> cyclic_matrix = datastructuretools.CyclicMatrix(
...     columns=[[0, 10, 20], [1, 11, 21], [2, 12, 22], [3, 13, 23]])
```

```
>>> cyclic_matrix
CyclicMatrix(3x4)
```

```
>>> cyclic_matrix[2]
CyclicTuple([20, 21, 22, 23])
```

```
>>> cyclic_matrix[2][2]
22
```

```
>>> cyclic_matrix[99]
CyclicTuple([0, 1, 2, 3])
```

```
>>> cyclic_matrix[99][99]
3
```

CyclicMatrix implements only item retrieval in this revision.

Concatenation and division remain to be implemented.

Standard transforms of linear algebra remain to be implemented.

## Bases

- `datastructuretools.Matrix`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`CyclicMatrix.columns`

Columns:

```
>>> cyclic_matrix = datastructuretools.CyclicMatrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> cyclic_matrix.columns
CyclicTuple([0, 10, 20], [1, 11, 21], [2, 12, 22], [3, 13, 23]))
```

Returns cyclic tuple.

`CyclicMatrix.rows`

Rows:

```
>>> cyclic_matrix = datastructuretools.CyclicMatrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> cyclic_matrix.rows
CyclicTuple([0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]))
```

Returns cyclic tuple.

## Special methods

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

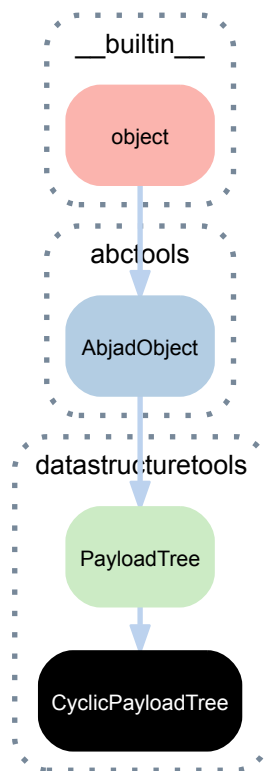
`CyclicMatrix.__getitem__(expr)`

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`CyclicMatrix.__repr__()`



### 51.2.4 datastructuretools.CyclicPayloadTree



**class** datastructuretools.**CyclicPayloadTree** (*expr*)

Abjad data structure to work with a sequence whose elements have been grouped into arbitrarily many levels of cyclic containment.

Exactly like the `PayloadTree` class but with the additional affordance that all integer indices of any size work at every level of structure.

Cyclic trees raise no index errors.

Here is a cyclic tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> cyclic_tree = datastructuretools.CyclicPayloadTree(sequence)
```

```
>>> cyclic_tree
CyclicPayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Here's an internal node:

```
>>> cyclic_tree[2]
CyclicPayloadTree([4, 5])
```

Here's the same node indexed with a different way:

```
>>> cyclic_tree[2]
CyclicPayloadTree([4, 5])
```

With a negative index:

```
>>> cyclic_tree[-2]
CyclicPayloadTree([4, 5])
```

And another negative index:

```
>>> cyclic_tree[-6]
CyclicPayloadTree([4, 5])
```

Here's a leaf node:

```
>>> cyclic_tree[2][0]
CyclicPayloadTree(4)
```

And here's the same node indexed a different way:

```
>>> cyclic_tree[2][20]
CyclicPayloadTree(4)
```

All other interface attributes function as in PayloadTree.

## Bases

- `datastructuretools.PayloadTree`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(PayloadTree).**children**

Children of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].children
(PayloadTree(2), PayloadTree(3))
```

Returns tuple of zero or more nodes.

(PayloadTree).**depth**

Depth of subtree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].depth
2
```

Returns nonnegative integer.

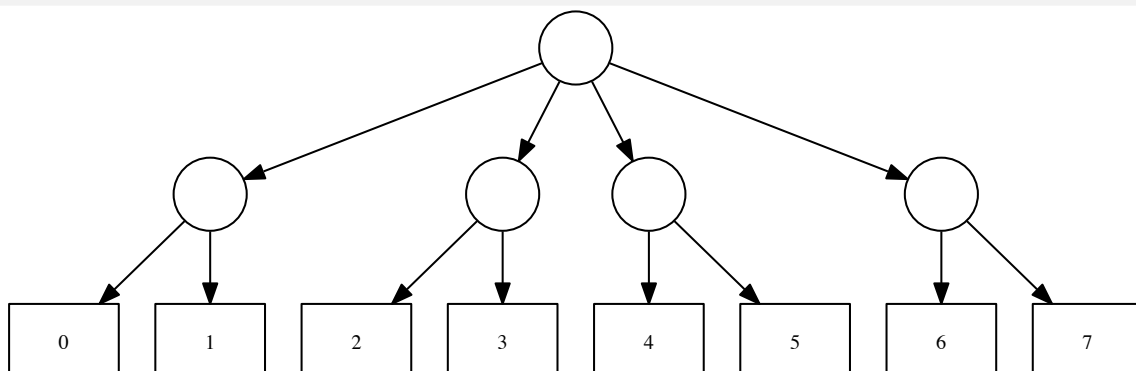
(PayloadTree).**graphviz\_format**

(PayloadTree).**graphviz\_graph**

The GraphvizGraph representation of the PayloadTree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> graph = tree.graphviz_graph
>>> iotools.graph(graph)
```



Returns graphviz graph.

`(PayloadTree).improper_parentage`  
Improper parentage of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].improper_parentage
(PayloadTree([2, 3]), PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]]))
```

Returns tuple of one or more nodes.

`(PayloadTree).index_in_parent`  
Index of node in parent of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].index_in_parent
1
```

Returns nonnegative integer.

`(PayloadTree).level`  
Level of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].level
1
```

Returns nonnegative integer.

`(PayloadTree).manifest_payload`  
Manifest payload of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree.manifest_payload
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> tree[-1].manifest_payload
[6, 7]
```

```
>>> tree[-1][-1].manifest_payload
[7]
```

Returns list.

`(PayloadTree).negative_level`  
Negative level of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].negative_level
-2
```

Returns negative integer.

`(PayloadTree).payload`  
Payload of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Returns none for interior node:

```
>>> tree.payload is None
True
```

```
>>> tree[-1].payload is None
True
```

Returns unwrapped payload for leaf node:

```
>>> tree[-1][-1].payload
7
```

Returns arbitrary expression or none.

(PayloadTree).**position**

Position of node relative to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].position
(1,)
```

Returns tuple of zero or more nonnegative integers.

(PayloadTree).**proper\_parentage**

Proper parentage of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].proper_parentage
(PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]]),)
```

Returns tuple of zero or more nodes.

(PayloadTree).**root**

Root of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].proper_parentage
(PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]]),)
```

Returns node.

CyclicPayloadTree.**storage\_format**

Storage format of cyclic tree.

Returns string.

(PayloadTree).**width**

Number of leaves in subtree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].width
2
```

Returns nonnegative integer.

## Methods

(PayloadTree).**get\_manifest\_payload\_of\_next\_n\_nodes\_at\_level** (*n*, *level*)

Get manifest payload of next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Get manifest payload of next 4 nodes at level 2:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(4, 2)
[1, 2, 3, 4]
```

Get manifest payload of next 3 nodes at level 1:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(3, 1)
[1, 2, 3, 4, 5]
```

Get manifest payload of next node at level 0:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(1, 0)
[1, 2, 3, 4, 5, 6, 7]
```

Get manifest payload of next 4 nodes at level -1:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(4, -1)
[1, 2, 3, 4]
```

Get manifest payload of next 3 nodes at level -2:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(3, -2)
[1, 2, 3, 4, 5]
```

Get manifest payload of previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-4, 2)
[6, 5, 4, 3]
```

Get manifest payload of previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-3, 1)
[6, 5, 4, 3, 2]
```

Get manifest payload of previous node at level 0:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-1, 0)
[6, 5, 4, 3, 2, 1, 0]
```

Get manifest payload of previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-4, -1)
[6, 5, 4, 3]
```

Get manifest payload of previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-3, -2)
[6, 5, 4, 3, 2]
```

Trim first node if necessary.

Returns list of arbitrary values.

(PayloadTree).**get\_next\_n\_complete\_nodes\_at\_level**(*n*, *level*)

Get next *n* complete nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Get next 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(4, 2)
[PayloadTree(1), PayloadTree(2), PayloadTree(3), PayloadTree(4)]
```

Get next 3 nodes at level 1:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(3, 1)
[PayloadTree([1]), PayloadTree([2, 3]), PayloadTree([4, 5]), PayloadTree([6, 7])]
```

Get next 4 nodes at level -1:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(4, -1)
[PayloadTree(1), PayloadTree(2), PayloadTree(3), PayloadTree(4)]
```

Get next 3 nodes at level -2:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(3, -2)
[PayloadTree([1]), PayloadTree([2, 3]), PayloadTree([4, 5]), PayloadTree([6, 7])]
```

Get previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-4, 2)
[PayloadTree(6), PayloadTree(5), PayloadTree(4), PayloadTree(3)]
```

Get previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-3, 1)
[PayloadTree([6]), PayloadTree([4, 5]), PayloadTree([2, 3]), PayloadTree([0, 1])]
```

Get previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-4, -1)
[PayloadTree(6), PayloadTree(5), PayloadTree(4), PayloadTree(3)]
```

Get previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-3, -2)
[PayloadTree([6]), PayloadTree([4, 5]), PayloadTree([2, 3]), PayloadTree([0, 1])]
```

Trim first node if necessary.

Returns list of nodes.

CyclicPayloadTree.**get\_next\_n\_nodes\_at\_level** (*n*, *level*)

Get next *n* nodes at *level*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.CyclicPayloadTree(sequence)
```

Get next 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_nodes_at_level(4, 2)
[CyclicPayloadTree(1), CyclicPayloadTree(2), CyclicPayloadTree(3), CyclicPayloadTree(4)]
```

Get next 10 nodes at level 2:

```
>>> for node in tree[0][0].get_next_n_nodes_at_level(10, 2):
...     node
CyclicPayloadTree(1)
CyclicPayloadTree(2)
CyclicPayloadTree(3)
CyclicPayloadTree(4)
CyclicPayloadTree(5)
CyclicPayloadTree(6)
CyclicPayloadTree(7)
CyclicPayloadTree(1)
CyclicPayloadTree(2)
CyclicPayloadTree(3)
```

### PREVIOUS MNODES ###

Get previous 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_nodes_at_level(-4, 2)
[CyclicPayloadTree(7), CyclicPayloadTree(6), CyclicPayloadTree(5), CyclicPayloadTree(4)]
```

Get previous 10 nodes at level 2:

```
>>> for node in tree[0][0].get_next_n_nodes_at_level(-10, 2):
...     node
CyclicPayloadTree(7)
CyclicPayloadTree(6)
CyclicPayloadTree(5)
CyclicPayloadTree(4)
CyclicPayloadTree(3)
CyclicPayloadTree(2)
CyclicPayloadTree(1)
CyclicPayloadTree(7)
CyclicPayloadTree(6)
CyclicPayloadTree(5)
```

Returns list of nodes.

`CyclicPayloadTree.get_node_at_position(position)`

Get node at *position*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> cyclic_tree = datastructuretools.CyclicPayloadTree(sequence)
```

```
>>> cyclic_tree.get_node_at_position((2, 1))
CyclicPayloadTree(5)
```

```
>>> cyclic_tree.get_node_at_position((2, 99))
CyclicPayloadTree(5)
```

```
>>> cyclic_tree.get_node_at_position((82, 1))
CyclicPayloadTree(5)
```

```
>>> cyclic_tree.get_node_at_position((82, 99))
CyclicPayloadTree(5)
```

Returns node.

`(PayloadTree).get_position_of_descendant(descendant)`

Get position of *descendent* relative to node rather than relative to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[3].get_position_of_descendant(tree[3][0])
(0,)
```

Returns tuple of zero or more nonnegative integers.

`(PayloadTree).index(node)`

Index of *node*:

```
>>> sequence = [0, 1, 2, 2, 3, 4]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> for node in tree:
...     node, tree.index(node)
(PayloadTree(0), 0)
(PayloadTree(1), 1)
(PayloadTree(2), 2)
(PayloadTree(2), 3)
(PayloadTree(3), 4)
(PayloadTree(4), 5)
```

Returns nonnegative integer.

`(PayloadTree).is_at_level(level)`

True when node is at *level* in containing tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1][1].is_at_level(-1)
True
```

False otherwise:

```
>>> tree[1][1].is_at_level(0)
False
```

Works for positive, negative and zero-valued *level*.

Returns boolean.

(PayloadTree) **.iterate\_at\_level** (*level*, *reverse=False*)  
Iterate tree at *level*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Left-to-right examples:

```
>>> for x in tree.iterate_at_level(0): x
...
PayloadTree([0, 1], [2, 3], [4, 5], [6, 7])
```

```
>>> for x in tree.iterate_at_level(1): x
...
PayloadTree([0, 1])
PayloadTree([2, 3])
PayloadTree([4, 5])
PayloadTree([6, 7])
```

```
>>> for x in tree.iterate_at_level(2): x
...
PayloadTree(0)
PayloadTree(1)
PayloadTree(2)
PayloadTree(3)
PayloadTree(4)
PayloadTree(5)
PayloadTree(6)
PayloadTree(7)
```

```
>>> for x in tree.iterate_at_level(-1): x
...
PayloadTree(0)
PayloadTree(1)
PayloadTree(2)
PayloadTree(3)
PayloadTree(4)
PayloadTree(5)
PayloadTree(6)
PayloadTree(7)
```

```
>>> for x in tree.iterate_at_level(-2): x
...
PayloadTree([0, 1])
PayloadTree([2, 3])
PayloadTree([4, 5])
PayloadTree([6, 7])
```

```
>>> for x in tree.iterate_at_level(-3): x
...
PayloadTree([0, 1], [2, 3], [4, 5], [6, 7])
```

Right-to-left examples:



```
>>> for x in tree.iterate_at_level(0, reverse=True): x
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> for x in tree.iterate_at_level(1, reverse=True): x
...
PayloadTree([6, 7])
PayloadTree([4, 5])
PayloadTree([2, 3])
PayloadTree([0, 1])
```

```
>>> for x in tree.iterate_at_level(2, reverse=True): x
...
PayloadTree(7)
PayloadTree(6)
PayloadTree(5)
PayloadTree(4)
PayloadTree(3)
PayloadTree(2)
PayloadTree(1)
PayloadTree(0)
```

```
>>> for x in tree.iterate_at_level(-1, reverse=True): x
...
PayloadTree(7)
PayloadTree(6)
PayloadTree(5)
PayloadTree(4)
PayloadTree(3)
PayloadTree(2)
PayloadTree(1)
PayloadTree(0)
```

```
>>> for x in tree.iterate_at_level(-2, reverse=True): x
...
PayloadTree([6, 7])
PayloadTree([4, 5])
PayloadTree([2, 3])
PayloadTree([0, 1])
```

```
>>> for x in tree.iterate_at_level(-3, reverse=True): x
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Returns node generator.

(PayloadTree).**iterate\_depth\_first** (*reverse=False*)  
Iterate tree depth-first:

**Example 1.** Iterate tree depth-first from left to right:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> for node in tree.iterate_depth_first(): node
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
PayloadTree([0, 1])
PayloadTree(0)
PayloadTree(1)
PayloadTree([2, 3])
PayloadTree(2)
PayloadTree(3)
PayloadTree([4, 5])
PayloadTree(4)
PayloadTree(5)
PayloadTree([6, 7])
PayloadTree(6)
PayloadTree(7)
```

**Example 2.** Iterate tree depth-first from right to left:

```
>>> for node in tree.iterate_depth_first(reverse=True): node
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
PayloadTree([6, 7])
PayloadTree(7)
PayloadTree(6)
PayloadTree([4, 5])
PayloadTree(5)
PayloadTree(4)
PayloadTree([2, 3])
PayloadTree(3)
PayloadTree(2)
PayloadTree([0, 1])
PayloadTree(1)
PayloadTree(0)
```

Returns node generator.

`CyclicPayloadTree.iterate_forever_depth_first` (*reverse=False*)  
Iterate tree depth first.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> cyclic_tree = datastructuretools.CyclicPayloadTree(sequence)
```

**Example 1.** Iterate from left to right:

```
>>> generator = cyclic_tree.iterate_forever_depth_first()
>>> for i in range(20):
...     generator.next()
CyclicPayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
CyclicPayloadTree([0, 1])
CyclicPayloadTree(0)
CyclicPayloadTree(1)
CyclicPayloadTree([2, 3])
CyclicPayloadTree(2)
CyclicPayloadTree(3)
CyclicPayloadTree([4, 5])
CyclicPayloadTree(4)
CyclicPayloadTree(5)
CyclicPayloadTree([6, 7])
CyclicPayloadTree(6)
CyclicPayloadTree(7)
CyclicPayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
CyclicPayloadTree([0, 1])
CyclicPayloadTree(0)
CyclicPayloadTree(1)
CyclicPayloadTree([2, 3])
CyclicPayloadTree(2)
CyclicPayloadTree(3)
```

**Example 2.** Iterate from right to left:

```
>>> generator = cyclic_tree.iterate_forever_depth_first(
...     reverse=True)
>>> for i in range(20):
...     generator.next()
CyclicPayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
CyclicPayloadTree([6, 7])
CyclicPayloadTree(7)
CyclicPayloadTree(6)
CyclicPayloadTree([4, 5])
CyclicPayloadTree(5)
CyclicPayloadTree(4)
CyclicPayloadTree([2, 3])
CyclicPayloadTree(3)
CyclicPayloadTree(2)
CyclicPayloadTree([0, 1])
CyclicPayloadTree(1)
CyclicPayloadTree(0)
CyclicPayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
CyclicPayloadTree([6, 7])
CyclicPayloadTree(7)
```

```
CyclicPayloadTree(6)
CyclicPayloadTree([4, 5])
CyclicPayloadTree(5)
CyclicPayloadTree(4)
```

Returns node generator.

(PayloadTree) **.iterate\_payload** (*reverse=False*)

Iterate tree payload:

**Example 1.** Iterate payload from left to right:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> for element in tree.iterate_payload():
...     element
...
0
1
2
3
4
5
6
7
```

**Example 2.** Iterate payload from right to left:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> for element in tree.iterate_payload(reverse=True):
...     element
...
7
6
5
4
3
2
1
0
```

Returns payload generator.

(PayloadTree) **.remove\_node** (*node*)

Remove *node* from tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree.remove_node(tree[1])
```

```
>>> tree
PayloadTree([[0, 1], [4, 5], [6, 7]])
```

Returns none.

(PayloadTree) **.remove\_to\_root** (*reverse=False*)

Remove node and all nodes left of node to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
```

```
>>> tree = datastructuretools.PayloadTree(sequence)
>>> tree[0][0].remove_to_root()
>>> tree
PayloadTree([[1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree = datastructuretools.PayloadTree(sequence)
>>> tree[0][1].remove_to_root()
>>> tree
PayloadTree([[2, 3], [4, 5], [6, 7]])
```

```
>>> tree = datastructuretools.PayloadTree(sequence)
>>> tree[1].remove_to_root()
>>> tree
PayloadTree([[4, 5], [6, 7]])
```

Modify in-place to root.

Returns none.

(PayloadTree).**to\_nested\_lists**()

Change tree to nested lists:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree.to_nested_lists()
[[0, 1], [2, 3], [4, 5], [6, 7]]
```

Returns list of lists.

## Special methods

(PayloadTree).**\_\_contains\_\_**(*expr*)

True when tree contains *expr*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[-1] in tree
True
```

Otherwise false:

```
>>> tree[-1][-1] in tree
False
```

Returns boolean.

(PayloadTree).**\_\_eq\_\_**(*expr*)

True when *expr* is the same type as tree and when the payload of all subtrees are equal:

```
>>> sequence_1 = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree_1 = datastructuretools.PayloadTree(sequence_1)
>>> sequence_2 = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree_2 = datastructuretools.PayloadTree(sequence_2)
>>> sequence_3 = [[0, 1], [2, 3], [4, 5]]
>>> tree_3 = datastructuretools.PayloadTree(sequence_3)
```

```
>>> tree_1 == tree_1
True
>>> tree_1 == tree_2
True
>>> tree_1 == tree_3
False
>>> tree_2 == tree_1
True
>>> tree_2 == tree_2
True
>>> tree_2 == tree_3
False
```

```
>>> tree_3 == tree_1
False
>>> tree_3 == tree_2
False
>>> tree_3 == tree_3
True
```

Returns boolean.

(PayloadTree).**\_\_getitem\_\_**(*expr*)

Get item from tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[-1]
PayloadTree([6, 7])
```

Get slice from tree:

```
>>> tree[-2:]
[PayloadTree([4, 5]), PayloadTree([6, 7])]
```

Returns node.

CyclicPayloadTree.**\_\_iter\_\_**()

(PayloadTree).**\_\_len\_\_**()

Returns the number of children in tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> len(tree)
4
```

Returns nonnegative integer.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(PayloadTree).**\_\_repr\_\_**()

Interpreter representation of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Returns string.

(PayloadTree).**\_\_str\_\_**()

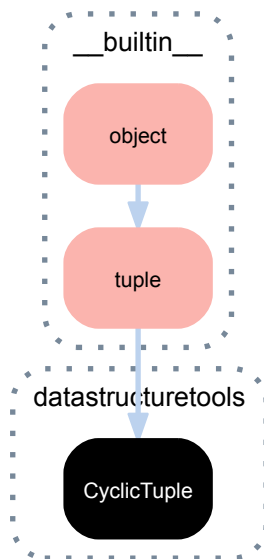
String representation of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> str(tree)
'[[0, 1], [2, 3], [4, 5], [6, 7]]'
```

Returns string.

### 51.2.5 `datastructuretools.CyclicTuple`



**class** `datastructuretools.CyclicTuple`  
 Abjad model of cyclic tuple:

```
>>> cyclic_tuple = datastructuretools.CyclicTuple('abcd')
```

```
>>> cyclic_tuple
CyclicTuple([a, b, c, d])
```

```
>>> for x in range(8):
...     print x, cyclic_tuple[x]
...
0 a
1 b
2 c
3 d
4 a
5 b
6 c
7 d
```

Cyclic tuples overload the item-getting method of built-in tuples.

Cyclic tuples return a value for any integer index.

Cyclic tuples otherwise behave exactly like built-in tuples.

#### Bases

- `__builtin__.tuple`
- `__builtin__.object`

#### Methods

`(tuple).count(value)` → integer – return number of occurrences of value

`(tuple).index(value[, start[, stop]])` → integer – return first index of value.  
 Raises `ValueError` if the value is not present.

## Special methods

```

(tuple).__add__()
    x.__add__(y) <==> x+y

(tuple).__contains__()
    x.__contains__(y) <==> y in x

(tuple).__eq__()
    x.__eq__(y) <==> x==y

(tuple).__ge__()
    x.__ge__(y) <==> x>=y

CyclicTuple.__getitem__(expr)

CyclicTuple.__getslice__(start_index, stop_index)

(tuple).__gt__()
    x.__gt__(y) <==> x>y

(tuple).__hash__() <==> hash(x)

(tuple).__iter__() <==> iter(x)

(tuple).__le__()
    x.__le__(y) <==> x<=y

(tuple).__len__() <==> len(x)

(tuple).__lt__()
    x.__lt__(y) <==> x<y

(tuple).__mul__()
    x.__mul__(n) <==> x*n

(tuple).__ne__()
    x.__ne__(y) <==> x!=y

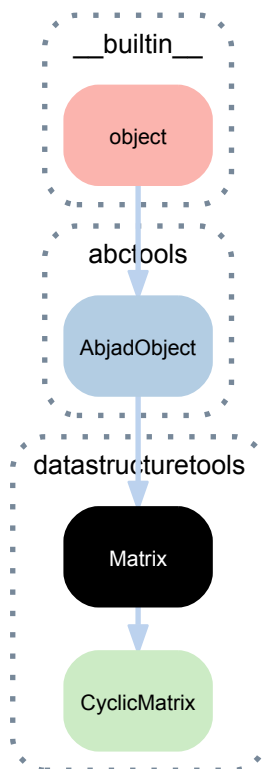
CyclicTuple.__repr__()

(tuple).__rmul__()
    x.__rmul__(n) <==> n*x

CyclicTuple.__str__()

```

### 51.2.6 datastructuretools.Matrix



**class** datastructuretools.**Matrix**(\*args, \*\*kwargs)  
 Abjad model of matrix.

Initialize from rows:

```
>>> matrix = datastructuretools.Matrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> matrix
Matrix(3x4)
```

```
>>> matrix[:]
((0, 1, 2, 3), (10, 11, 12, 13), (20, 21, 22, 23))
```

```
>>> matrix[2]
(20, 21, 22, 23)
```

```
>>> matrix[2][0]
20
```

Initialize from columns:

```
>>> matrix = datastructuretools.Matrix(
...     columns=[[0, 10, 20], [1, 11, 21], [2, 12, 22], [3, 13, 23]])
```

```
>>> matrix
Matrix(3x4)
```

```
>>> matrix[:]
((0, 1, 2, 3), (10, 11, 12, 13), (20, 21, 22, 23))
```

```
>>> matrix[2]
(20, 21, 22, 23)
```

```
>>> matrix[2][0]
20
```



Matrix implements only item retrieval in this revision.

Concatenation and division remain to be implemented.

Standard transforms of linear algebra remain to be implemented.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### Matrix.columns

Columns:

```
>>> matrix = datastructuretools.Matrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> matrix.columns
((0, 10, 20), (1, 11, 21), (2, 12, 22), (3, 13, 23))
```

Returns tuple.

### Matrix.rows

Rows:

```
>>> matrix = datastructuretools.Matrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> matrix.rows
((0, 1, 2, 3), (10, 11, 12, 13), (20, 21, 22, 23))
```

Returns tuple.

## Special methods

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

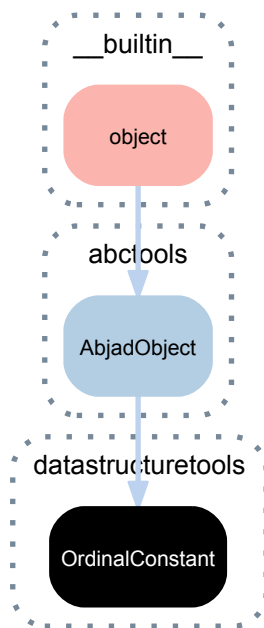
Matrix.**\_\_getitem\_\_**(*expr*)

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

Matrix.**\_\_repr\_\_**()

51.2.7 `datastructuretools.OrdinalConstant`

**class** `datastructuretools.OrdinalConstant`

Ordinal constant.

Initialize with *dimension*, *value* and *representation*:

```
>>> Left = datastructuretools.OrdinalConstant('x', -1, 'Left')
>>> Left
Left
```

```
>>> Right = datastructuretools.OrdinalConstant('x', 1, 'Right')
>>> Right
Right
```

```
>>> Left < Right
True
```

Comparing like-dimensioned ordinal constants is allowed:

```
>>> Up = datastructuretools.OrdinalConstant('y', 1, 'Up')
>>> Up
Up
```

```
>>> Down = datastructuretools.OrdinalConstant('y', -1, 'Down')
>>> Down
Down
```

```
>>> Down < Up
True
```

Comparing differently dimensioned ordinal constants raises an exception:

```
>>> import py.test
```

```
>>> bool(py.test.raises(Exception, 'Left < Up'))
True
```

The `Left`, `Right`, `Center`, `Up` and `Down` constants shown here load into Python's built-in namespace on `Abjad` import.

These four objects can be used as constant values supplied to keywords.

This behavior is similar to `True`, `False` and `None`.

Ordinal constants are immutable.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

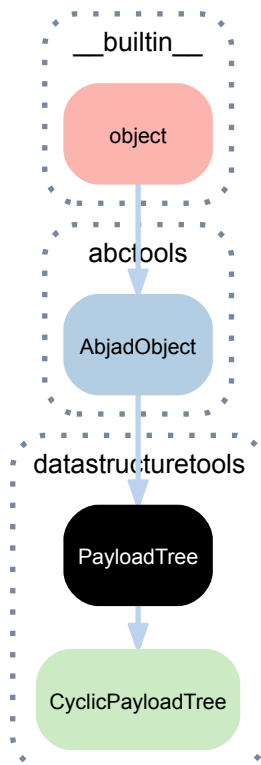
`OrdinalConstant.storage_format`  
Storage format of ordinal constant.  
Returns string.

## Special methods

`OrdinalConstant.__eq__(expr)`  
`OrdinalConstant.__ge__(expr)`  
`OrdinalConstant.__gt__(expr)`  
`OrdinalConstant.__le__(expr)`  
`OrdinalConstant.__lt__(expr)`  
`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`OrdinalConstant.__new__(dimension, value, representation)`  
`OrdinalConstant.__repr__()`

## 51.2.8 datastructuretools.PayloadTree



**class** `datastructuretools.PayloadTree` (*expr*)

Abjad data structure to work with a sequence whose elements have been grouped into arbitrarily many levels of containment.

Here is a tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree.parent is None
True
```

```
>>> tree.children
(PayloadTree([0, 1]), PayloadTree([2, 3]), PayloadTree([4, 5]), PayloadTree([6, 7]))
```

```
>>> tree.depth
3
```

Here's an internal node:

```
>>> tree[2]
PayloadTree([4, 5])
```

```
>>> tree[2].parent
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree[2].children
(PayloadTree(4), PayloadTree(5))
```

```
>>> tree[2].depth
2
```

```
>>> tree[2].level
1
```

Here's a leaf node:

```
>>> tree[2][0]
PayloadTree(4)
```

```
>>> tree[2][0].parent
PayloadTree([4, 5])
```

```
>>> tree[2][0].children
()
```

```
>>> tree[2][0].depth
1
```

```
>>> tree[2][0].level
2
```

```
>>> tree[2][0].position
(2, 0)
```

```
>>> tree[2][0].payload
4
```

Only leaf nodes carry payload. Internal nodes carry no payload.

Negative levels are available to work with trees bottom-up instead of top-down.

Trees do not yet implement append or extend methods.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `PayloadTree.children`

Children of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].children
(PayloadTree(2), PayloadTree(3))
```

Returns tuple of zero or more nodes.

### `PayloadTree.depth`

Depth of subtree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].depth
2
```

Returns nonnegative integer.

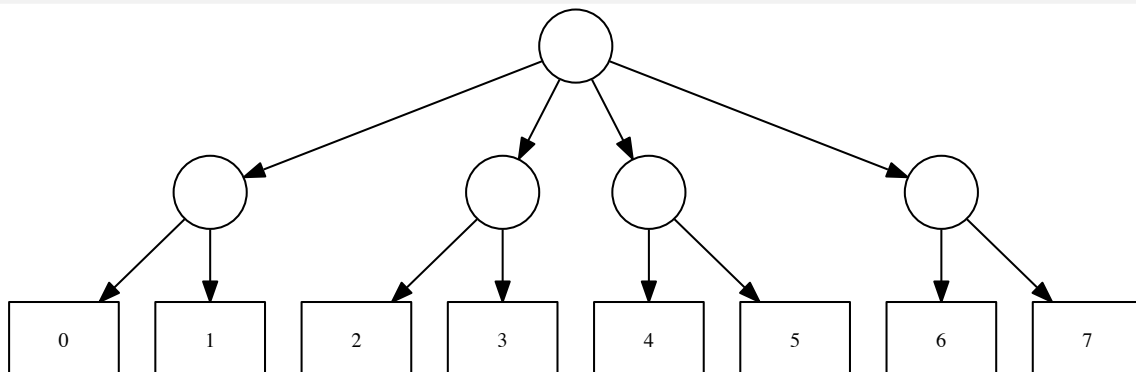
### `PayloadTree.graphviz_format`

### `PayloadTree.graphviz_graph`

The GraphvizGraph representation of the PayloadTree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> graph = tree.graphviz_graph
>>> iotools.graph(graph)
```



Returns graphviz graph.

### `PayloadTree.improper_parentage`

Improper parentage of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].improper_parentage
(PayloadTree([2, 3]), PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]]))
```

Returns tuple of one or more nodes.

**PayloadTree.index\_in\_parent**

Index of node in parent of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].index_in_parent
1
```

Returns nonnegative integer.

**PayloadTree.level**

Level of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].level
1
```

Returns nonnegative integer.

**PayloadTree.manifest\_payload**

Manifest payload of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree.manifest_payload
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> tree[-1].manifest_payload
[6, 7]
```

```
>>> tree[-1][-1].manifest_payload
[7]
```

Returns list.

**PayloadTree.negative\_level**

Negative level of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].negative_level
-2
```

Returns negative integer.

**PayloadTree.payload**

Payload of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Returns none for interior node:

```
>>> tree.payload is None
True
```

```
>>> tree[-1].payload is None
True
```

Returns unwrapped payload for leaf node:

```
>>> tree[-1][-1].payload
7
```

Returns arbitrary expression or none.

**PayloadTree.position**

Position of node relative to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].position
(1,)
```

Returns tuple of zero or more nonnegative integers.

**PayloadTree.proper\_parentage**

Proper parentage of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].proper_parentage
(PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]]),)
```

Returns tuple of zero or more nodes.

**PayloadTree.root**

Root of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].proper_parentage
(PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]]),)
```

Returns node.

**PayloadTree.storage\_format**

PayloadTree storage format:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> print tree.storage_format
datastructuretools.PayloadTree(
  [[0, 1], [2, 3], [4, 5], [6, 7]]
)
```

Returns string.

**PayloadTree.width**

Number of leaves in subtree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].width
2
```

Returns nonnegative integer.

**Methods****PayloadTree.get\_manifest\_payload\_of\_next\_n\_nodes\_at\_level** (*n*, *level*)Get manifest payload of next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Get manifest payload of next 4 nodes at level 2:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(4, 2)
[1, 2, 3, 4]
```

Get manifest payload of next 3 nodes at level 1:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(3, 1)
[1, 2, 3, 4, 5]
```

Get manifest payload of next node at level 0:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(1, 0)
[1, 2, 3, 4, 5, 6, 7]
```

Get manifest payload of next 4 nodes at level -1:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(4, -1)
[1, 2, 3, 4]
```

Get manifest payload of next 3 nodes at level -2:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(3, -2)
[1, 2, 3, 4, 5]
```

Get manifest payload of previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-4, 2)
[6, 5, 4, 3]
```

Get manifest payload of previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-3, 1)
[6, 5, 4, 3, 2]
```

Get manifest payload of previous node at level 0:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-1, 0)
[6, 5, 4, 3, 2, 1, 0]
```

Get manifest payload of previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-4, -1)
[6, 5, 4, 3]
```

Get manifest payload of previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-3, -2)
[6, 5, 4, 3, 2]
```

Trim first node if necessary.

Returns list of arbitrary values.

`PayloadTree.get_next_n_complete_nodes_at_level(n, level)`

Get next *n* complete nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Get next 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(4, 2)
[PayloadTree(1), PayloadTree(2), PayloadTree(3), PayloadTree(4)]
```

Get next 3 nodes at level 1:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(3, 1)
[PayloadTree([1]), PayloadTree([2, 3]), PayloadTree([4, 5]), PayloadTree([6, 7])]
```

Get next 4 nodes at level -1:



```
>>> tree[0][0].get_next_n_complete_nodes_at_level(4, -1)
[PayloadTree(1), PayloadTree(2), PayloadTree(3), PayloadTree(4)]
```

Get next 3 nodes at level -2:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(3, -2)
[PayloadTree([1]), PayloadTree([2, 3]), PayloadTree([4, 5]), PayloadTree([6, 7])]
```

Get previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-4, 2)
[PayloadTree(6), PayloadTree(5), PayloadTree(4), PayloadTree(3)]
```

Get previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-3, 1)
[PayloadTree([6]), PayloadTree([4, 5]), PayloadTree([2, 3]), PayloadTree([0, 1])]
```

Get previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-4, -1)
[PayloadTree(6), PayloadTree(5), PayloadTree(4), PayloadTree(3)]
```

Get previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-3, -2)
[PayloadTree([6]), PayloadTree([4, 5]), PayloadTree([2, 3]), PayloadTree([0, 1])]
```

Trim first node if necessary.

Returns list of nodes.

`PayloadTree.get_next_n_nodes_at_level(n, level)`

Get next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Get next 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_nodes_at_level(4, 2)
[PayloadTree(1), PayloadTree(2), PayloadTree(3), PayloadTree(4)]
```

Get next 3 nodes at level 1:

```
>>> tree[0][0].get_next_n_nodes_at_level(3, 1)
[PayloadTree([1]), PayloadTree([2, 3]), PayloadTree([4, 5])]
```

Get next node at level 0:

```
>>> tree[0][0].get_next_n_nodes_at_level(1, 0)
[PayloadTree([1], [2, 3], [4, 5], [6, 7])]
```

Get next 4 nodes at level -1:

```
>>> tree[0][0].get_next_n_nodes_at_level(4, -1)
[PayloadTree(1), PayloadTree(2), PayloadTree(3), PayloadTree(4)]
```

Get next 3 nodes at level -2:

```
>>> tree[0][0].get_next_n_nodes_at_level(3, -2)
[PayloadTree([1]), PayloadTree([2, 3]), PayloadTree([4, 5])]
```

Get previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-4, 2)
[PayloadTree(6), PayloadTree(5), PayloadTree(4), PayloadTree(3)]
```

Get previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-3, 1)
[PayloadTree([6]), PayloadTree([4, 5]), PayloadTree([2, 3])]
```

Get previous node at level 0:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-1, 0)
[PayloadTree([[0, 1], [2, 3], [4, 5], [6]])]
```

Get previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-4, -1)
[PayloadTree(6), PayloadTree(5), PayloadTree(4), PayloadTree(3)]
```

Get previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-3, -2)
[PayloadTree([6]), PayloadTree([4, 5]), PayloadTree([2, 3])]
```

Trim first node if necessary.

Returns list of nodes.

`PayloadTree.get_node_at_position(position)`

Get node at *position*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree.get_node_at_position((2, 1))
PayloadTree(5)
```

Returns node.

`PayloadTree.get_position_of_descendant(descendant)`

Get position of *descendant* relative to node rather than relative to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[3].get_position_of_descendant(tree[3][0])
(0,)
```

Returns tuple of zero or more nonnegative integers.

`PayloadTree.index(node)`

Index of *node*:

```
>>> sequence = [0, 1, 2, 2, 3, 4]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> for node in tree:
...     node, tree.index(node)
(PayloadTree(0), 0)
(PayloadTree(1), 1)
(PayloadTree(2), 2)
(PayloadTree(2), 3)
(PayloadTree(3), 4)
(PayloadTree(4), 5)
```

Returns nonnegative integer.

`PayloadTree.is_at_level(level)`

True when node is at *level* in containing tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1][1].is_at_level(-1)
True
```

False otherwise:

```
>>> tree[1][1].is_at_level(0)
False
```

Works for positive, negative and zero-valued *level*.

Returns boolean.

`PayloadTree.iterate_at_level` (*level*, *reverse=False*)  
Iterate tree at *level*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Left-to-right examples:

```
>>> for x in tree.iterate_at_level(0): x
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> for x in tree.iterate_at_level(1): x
...
PayloadTree([0, 1])
PayloadTree([2, 3])
PayloadTree([4, 5])
PayloadTree([6, 7])
```

```
>>> for x in tree.iterate_at_level(2): x
...
PayloadTree(0)
PayloadTree(1)
PayloadTree(2)
PayloadTree(3)
PayloadTree(4)
PayloadTree(5)
PayloadTree(6)
PayloadTree(7)
```

```
>>> for x in tree.iterate_at_level(-1): x
...
PayloadTree(0)
PayloadTree(1)
PayloadTree(2)
PayloadTree(3)
PayloadTree(4)
PayloadTree(5)
PayloadTree(6)
PayloadTree(7)
```

```
>>> for x in tree.iterate_at_level(-2): x
...
PayloadTree([0, 1])
PayloadTree([2, 3])
PayloadTree([4, 5])
PayloadTree([6, 7])
```

```
>>> for x in tree.iterate_at_level(-3): x
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Right-to-left examples:

```
>>> for x in tree.iterate_at_level(0, reverse=True): x
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> for x in tree.iterate_at_level(1, reverse=True): x
...
PayloadTree([6, 7])
```

```
PayloadTree([4, 5])
PayloadTree([2, 3])
PayloadTree([0, 1])
```

```
>>> for x in tree.iterate_at_level(2, reverse=True): x
...
PayloadTree(7)
PayloadTree(6)
PayloadTree(5)
PayloadTree(4)
PayloadTree(3)
PayloadTree(2)
PayloadTree(1)
PayloadTree(0)
```

```
>>> for x in tree.iterate_at_level(-1, reverse=True): x
...
PayloadTree(7)
PayloadTree(6)
PayloadTree(5)
PayloadTree(4)
PayloadTree(3)
PayloadTree(2)
PayloadTree(1)
PayloadTree(0)
```

```
>>> for x in tree.iterate_at_level(-2, reverse=True): x
...
PayloadTree([6, 7])
PayloadTree([4, 5])
PayloadTree([2, 3])
PayloadTree([0, 1])
```

```
>>> for x in tree.iterate_at_level(-3, reverse=True): x
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Returns node generator.

`PayloadTree.iterate_depth_first` (*reverse=False*)

Iterate tree depth-first:

**Example 1.** Iterate tree depth-first from left to right:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> for node in tree.iterate_depth_first(): node
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
PayloadTree([0, 1])
PayloadTree(0)
PayloadTree(1)
PayloadTree([2, 3])
PayloadTree(2)
PayloadTree(3)
PayloadTree([4, 5])
PayloadTree(4)
PayloadTree(5)
PayloadTree([6, 7])
PayloadTree(6)
PayloadTree(7)
```

**Example 2.** Iterate tree depth-first from right to left:

```
>>> for node in tree.iterate_depth_first(reverse=True): node
...
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
PayloadTree([6, 7])
PayloadTree(7)
PayloadTree(6)
```

```

PayloadTree([4, 5])
PayloadTree(5)
PayloadTree(4)
PayloadTree([2, 3])
PayloadTree(3)
PayloadTree(2)
PayloadTree([0, 1])
PayloadTree(1)
PayloadTree(0)

```

Returns node generator.

`PayloadTree.iterate_payload` (*reverse=False*)  
Iterate tree payload:

**Example 1.** Iterate payload from left to right:

```

>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> for element in tree.iterate_payload():
...     element
...
0
1
2
3
4
5
6
7

```

**Example 2.** Iterate payload from right to left:

```

>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> for element in tree.iterate_payload(reverse=True):
...     element
...
7
6
5
4
3
2
1
0

```

Returns payload generator.

`PayloadTree.remove_node` (*node*)  
Remove *node* from tree:

```

>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> tree.remove_node(tree[1])

```

```

>>> tree
PayloadTree([[0, 1], [4, 5], [6, 7]])

```

Returns none.

`PayloadTree.remove_to_root` (*reverse=False*)  
Remove node and all nodes left of node to root:

```

>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]

```

```
>>> tree = datastructuretools.PayloadTree(sequence)
>>> tree[0][0].remove_to_root()
>>> tree
PayloadTree([[1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree = datastructuretools.PayloadTree(sequence)
>>> tree[0][1].remove_to_root()
>>> tree
PayloadTree([[2, 3], [4, 5], [6, 7]])
```

```
>>> tree = datastructuretools.PayloadTree(sequence)
>>> tree[1].remove_to_root()
>>> tree
PayloadTree([[4, 5], [6, 7]])
```

Modify in-place to root.

Returns none.

`PayloadTree.to_nested_lists()`

Change tree to nested lists:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree.to_nested_lists()
[[0, 1], [2, 3], [4, 5], [6, 7]]
```

Returns list of lists.

## Special methods

`PayloadTree.__contains__(expr)`

True when tree contains *expr*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[-1] in tree
True
```

Otherwise false:

```
>>> tree[-1][-1] in tree
False
```

Returns boolean.

`PayloadTree.__eq__(expr)`

True when *expr* is the same type as tree and when the payload of all subtrees are equal:

```
>>> sequence_1 = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree_1 = datastructuretools.PayloadTree(sequence_1)
>>> sequence_2 = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree_2 = datastructuretools.PayloadTree(sequence_2)
>>> sequence_3 = [[0, 1], [2, 3], [4, 5]]
>>> tree_3 = datastructuretools.PayloadTree(sequence_3)
```

```
>>> tree_1 == tree_1
True
>>> tree_1 == tree_2
True
>>> tree_1 == tree_3
False
>>> tree_2 == tree_1
```

```

True
>>> tree_2 == tree_2
True
>>> tree_2 == tree_3
False
>>> tree_3 == tree_1
False
>>> tree_3 == tree_2
False
>>> tree_3 == tree_3
True

```

Returns boolean.

`PayloadTree.__getitem__(expr)`

Get item from tree:

```

>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> tree[-1]
PayloadTree([6, 7])

```

Get slice from tree:

```

>>> tree[-2:]
[PayloadTree([4, 5]), PayloadTree([6, 7])]

```

Returns node.

`PayloadTree.__len__()`

Returns the number of children in tree:

```

>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> len(tree)
4

```

Returns nonnegative integer.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`PayloadTree.__repr__()`

Interpreter representation of tree:

```

>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> tree
PayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])

```

Returns string.

`PayloadTree.__str__()`

String representation of tree:

```

>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

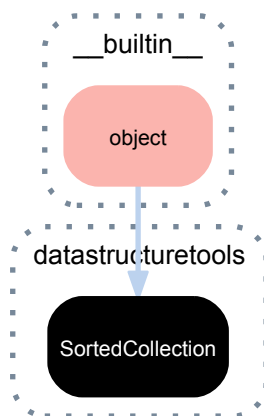
```

>>> str(tree)
'[[0, 1], [2, 3], [4, 5], [6, 7]]'

```

Returns string.

### 51.2.9 datastructuretools.SortedCollection



**class** datastructuretools.SortedCollection (iterable=(), key=None)  
Sequence sorted by a key function.

SortedCollection() is much easier to work with than using bisect() directly. It supports key functions like those use in sorted(), min(), and max(). The result of the key function call is saved so that keys can be searched efficiently.

Instead of returning an insertion-point which can be hard to interpret, the five find-methods return a specific item in the sequence. They can scan for exact matches, the last item less-than-or-equal to a key, or the first item greater-than-or-equal to a key.

Once found, an item's ordinal position can be located with the index() method. New items can be added with the insert() and insert\_right() methods. Old items can be deleted with the remove() method.

The usual sequence methods are provided to support indexing, slicing, length lookup, clearing, copying, forward and reverse iteration, contains checking, item counts, item removal, and a nice looking repr.

Finding and indexing are  $O(\log n)$  operations while iteration and insertion are  $O(n)$ . The initial sort is  $O(n \log n)$ .

The key function is stored in the 'key' attribute for easy introspection or so that you can assign a new key function (triggering an automatic re-sort).

In short, the class was designed to handle all of the common use cases for bisect but with a simpler API and support for key functions.

```
>>> from pprint import pprint
>>> from operator import itemgetter
```

```
>>> s = datastructuretools.SortedCollection(key=itemgetter(2))
>>> for record in [
...     ('roger', 'young', 30),
...     ('angela', 'jones', 28),
...     ('bill', 'smith', 22),
...     ('david', 'thomas', 32)]:
...     s.insert(record)
```

```
>>> pprint(list(s))           # show records sorted by age
[('bill', 'smith', 22),
 ('angela', 'jones', 28),
 ('roger', 'young', 30),
 ('david', 'thomas', 32)]
```

```
>>> s.find_le(29)             # find oldest person aged 29 or younger
('angela', 'jones', 28)
>>> s.find_lt(28)            # find oldest person under 28
('bill', 'smith', 22)
>>> s.find_gt(28)            # find youngest person over 28
('roger', 'young', 30)
```



```

>>> r = s.find_ge(32)      # find youngest person aged 32 or older
>>> s.index(r)             # get the index of their record
3
>>> s[3]                  # fetch the record at that index
('david', 'thomas', 32)

>>> s.key = itemgetter(0)  # now sort by first name
>>> pprint(list(s))
[('angela', 'jones', 28),
 ('bill', 'smith', 22),
 ('david', 'thomas', 32),
 ('roger', 'young', 30)]

```

## Bases

- `__builtin__.object`

## Read/write properties

`SortedCollection.key`  
key function

## Methods

`SortedCollection.clear()`

`SortedCollection.copy()`

`SortedCollection.count(item)`  
Returns number of occurrences of item

`SortedCollection.find(k)`  
Returns first item with a key == k. Raise `ValueError` if not found.

`SortedCollection.find_ge(k)`  
Returns first item with a key >= equal to k. Raise `ValueError` if not found

`SortedCollection.find_gt(k)`  
Returns first item with a key > k. Raise `ValueError` if not found

`SortedCollection.find_le(k)`  
Returns last item with a key <= k. Raise `ValueError` if not found.

`SortedCollection.find_lt(k)`  
Returns last item with a key < k. Raise `ValueError` if not found.

`SortedCollection.index(item)`  
Find the position of an item. Raise `ValueError` if not found.

`SortedCollection.insert(item)`  
Insert a new item. If equal keys are found, add to the left

`SortedCollection.insert_right(item)`  
Insert a new item. If equal keys are found, add to the right

`SortedCollection.remove(item)`  
Remove first occurrence of item. Raise `ValueError` if not found

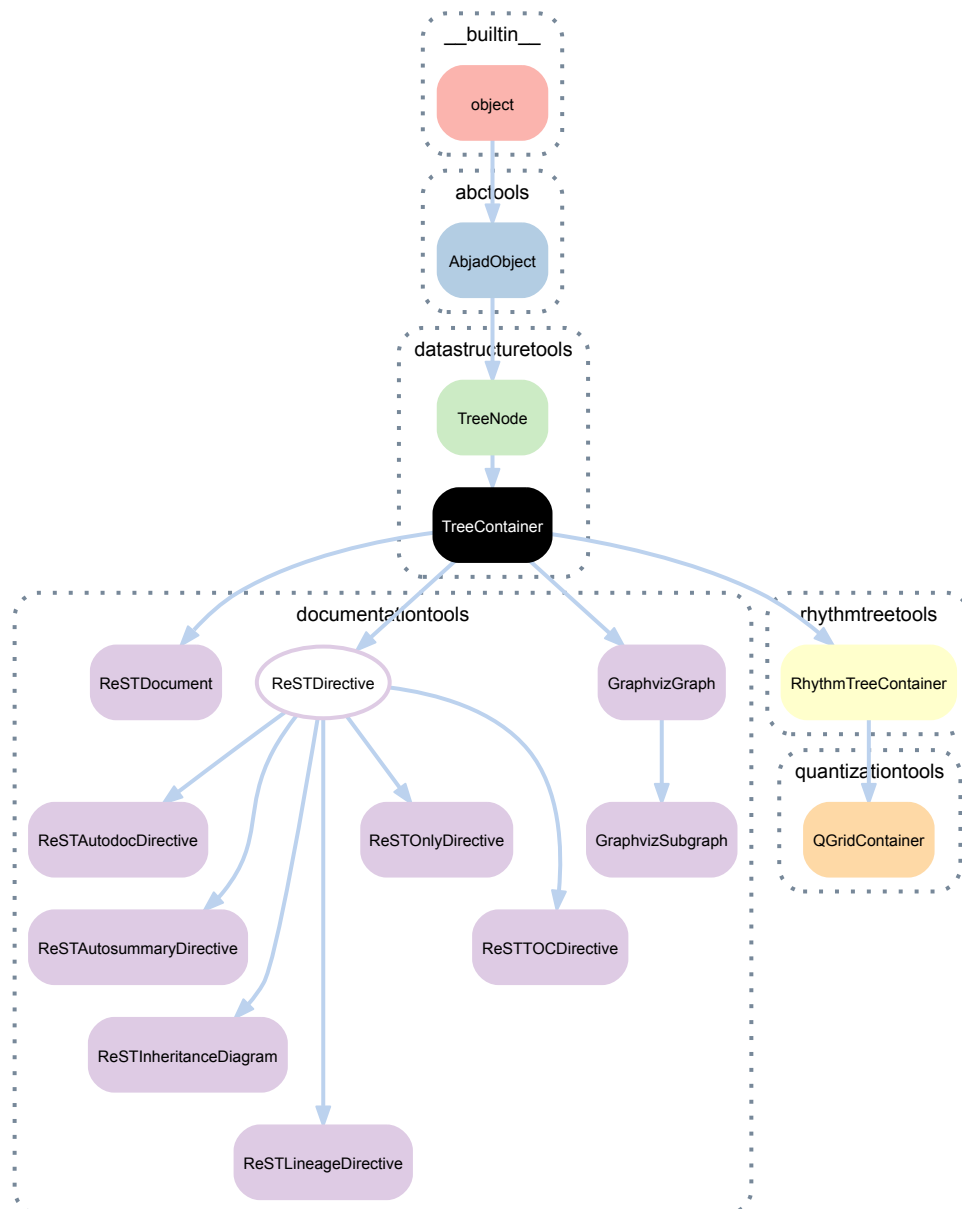
## Special methods

`SortedCollection.__contains__(item)`

`SortedCollection.__getitem__(i)`

```
SortedCollection.__iter__()
SortedCollection.__len__()
SortedCollection.__repr__()
SortedCollection.__reversed__()
```

## 51.2.10 datastructuretools.TreeContainer



**class** datastructuretools.**TreeContainer** (*children=None, name=None*)

An inner node in a generalized tree data-structure:

```
>>> a = datastructuretools.TreeContainer()
>>> a
TreeContainer()
```

```
>>> b = datastructuretools.TreeNode()
>>> a.append(b)
>>> a
TreeContainer(
  children=(
```

```

        TreeNode(),
    )
)

```

Return *TreeContainer* instance.

## Bases

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### `TreeContainer.children`

The children of a *TreeContainer* instance:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()

```

```

>>> a.extend([b, c])
>>> b.extend([d, e])

```

```

>>> a.children == (b, c)
True

```

```

>>> b.children == (d, e)
True

```

```

>>> e.children == ()
True

```

Returns tuple of *TreeNode* instances.

### `(TreeNode).depth`

The depth of a node in a rhythm-tree structure:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)

```

```

>>> a.depth
0

```

```

>>> a[0].depth
1

```

```

>>> a[0][0].depth
2

```

Returns int.

### `(TreeNode).depthwise_inventory`

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```

>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')

```

```
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

(TreeNode).**graph\_order**

(TreeNode).**improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

TreeContainer.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

`TreeContainer.nodes`

The collection of *TreeNode*s produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

`(TreeNode).parent`

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`(TreeNode).proper_parentage`

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

(*TreeNode*) **.name**

## Methods

*TreeContainer*.**.append**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

`TreeContainer.extend(expr)`

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

`TreeContainer.index(node)`

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

`TreeContainer.insert(i, node)`

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

`TreeContainer.pop(i=-1)`

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

`TreeContainer.remove(node)`

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

`TreeContainer.__contains__(expr)`

True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```



```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

```
(TreeNode) .__copy__ (*args)
```

```
(TreeNode) .__deepcopy__ (*args)
```

```
TreeContainer.__delitem__ (i)
```

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

```
TreeContainer.__eq__ (expr)
```

True if type, duration and children are equivalent, otherwise False.

Returns boolean.

```
TreeContainer.__getitem__ (i)
```

Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(TreeNode) .\_\_getstate\_\_()

TreeContainer.\_\_iter\_\_()

TreeContainer.\_\_len\_\_()

Returns nonnegative integer number of nodes in container.

(TreeNode) .\_\_ne\_\_(*expr*)

(TreeNode) .\_\_repr\_\_()

TreeContainer.\_\_setitem\_\_(*i*, *expr*)

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

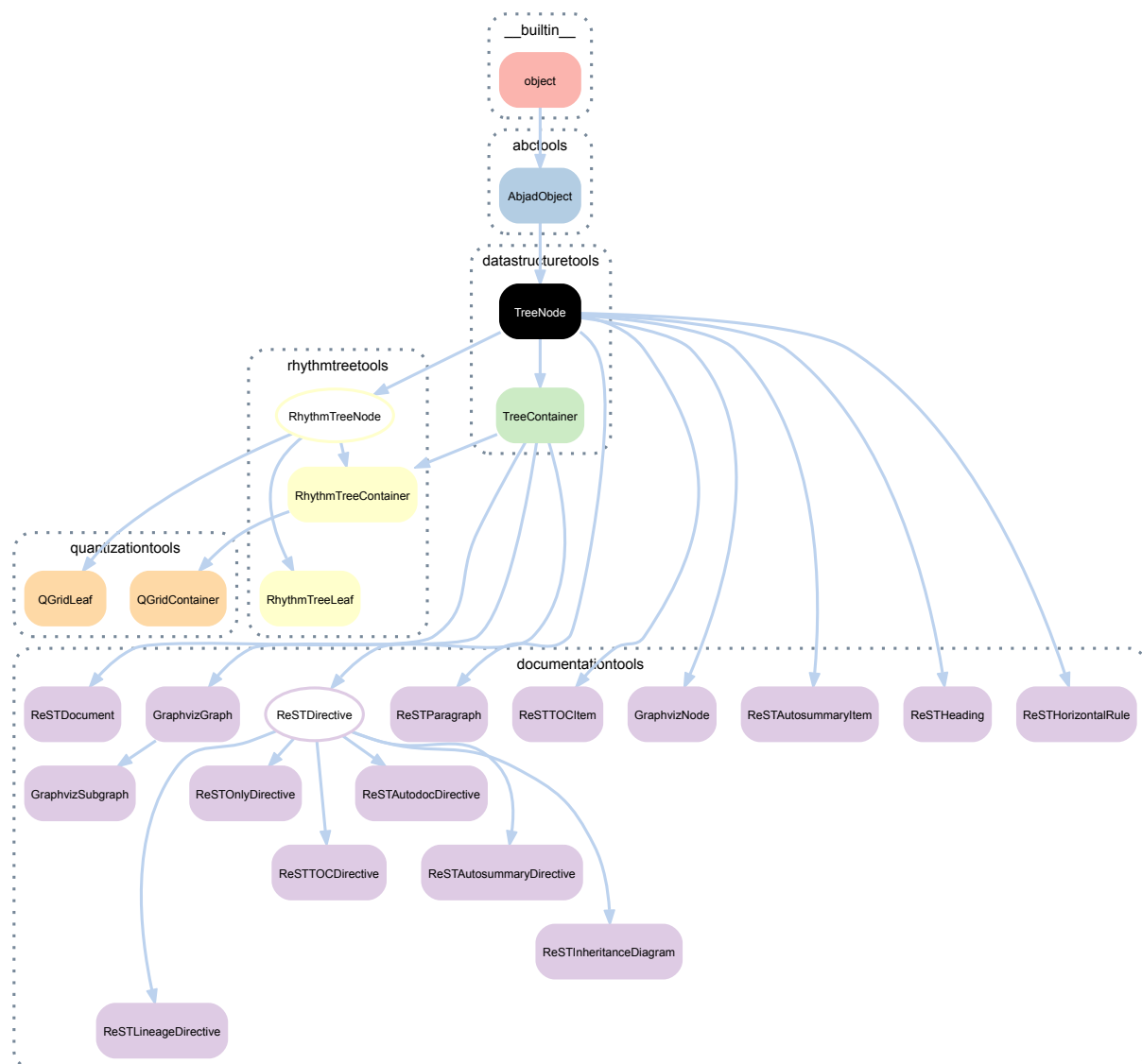
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

(TreeNode) .\_\_setstate\_\_(*state*)

### 51.2.11 datastructuretools.TreeNode



**class** datastructuretools.**TreeNode** (*name=None*)

A node in a generalized tree.

Return *TreeNode* instance.

#### Bases

- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

**TreeNode.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

#### TreeNode.**depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

#### TreeNode.**graph\_order**

#### TreeNode.**improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

**TreeNode.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

**TreeNode.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

**TreeNode.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

**Read/write properties****TreeNode.name**

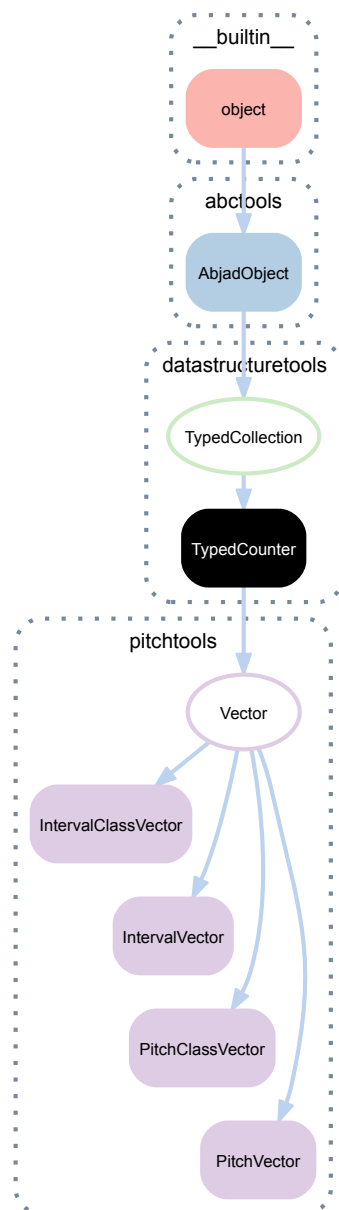
## Special methods

```

TreeNode.__copy__ (*args)
TreeNode.__deepcopy__ (*args)
TreeNode.__eq__ (expr)
TreeNode.__getstate__ ()
TreeNode.__ne__ (expr)
TreeNode.__repr__ ()
TreeNode.__setstate__ (state)

```

### 51.2.12 datastructuretools.TypedCounter



```

class datastructuretools.TypedCounter (tokens=None,    item_class=None,    name=None,
                                     **kwargs)

```

## Bases

- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

## Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

## Methods

`TypedCounter.clear()`

`TypedCounter.copy()`

`TypedCounter.elements()`

`TypedCounter.items()`

`TypedCounter.iteritems()`

`TypedCounter.iterkeys()`

`TypedCounter.itervalues()`

`TypedCounter.keys()`

`TypedCounter.most_common(n=None)`

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

`TypedCounter.subtract(iterable=None, **kwargs)`

`TypedCounter.update(iterable=None, **kwargs)`

`TypedCounter.values()`

`TypedCounter.viewitems()`

`TypedCounter.viewkeys()`

`TypedCounter.viewvalues()`

## Special methods

`TypedCounter.__add__(expr)`

`TypedCounter.__and__(expr)`

`(TypedCollection).__contains__(token)`

`TypedCounter.__delitem__(token)`

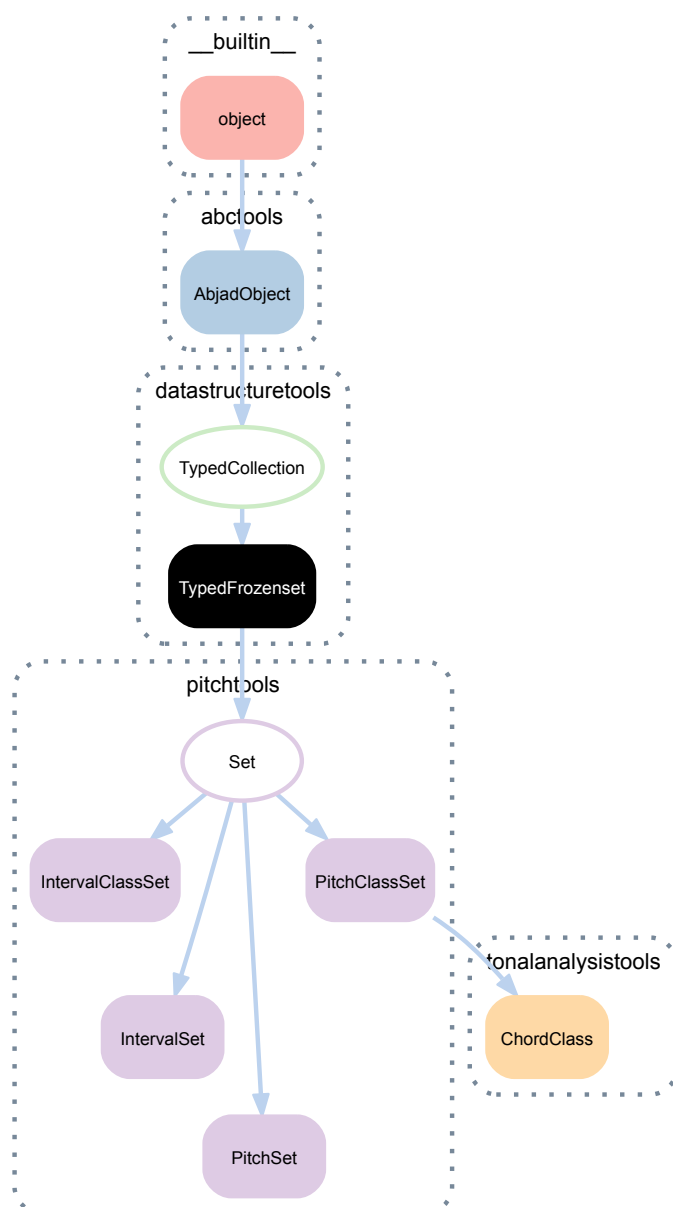
`(TypedCollection).__eq__(expr)`

```

TypedCounter.__getitem__(token)
(TypedCollection).__iter__()
(TypedCollection).__len__()
TypedCounter.__missing__(token)
(TypedCollection).__ne__(expr)
TypedCounter.__or__(expr)
(AbjadObject).__repr__()
    Interpreter representation of Abjad object.
    Returns string.
TypedCounter.__setitem__(token, value)
TypedCounter.__sub__(expr)

```

### 51.2.13 datastructuretools.TypedFrozenset





**class** `datastructuretools.TypedFrozenset` (*tokens=None, item\_class=None, name=None*)

### Bases

- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`(TypedCollection).item_class`  
Item class to coerce tokens into.

`(TypedCollection).storage_format`  
Storage format of typed tuple.

### Read/write properties

`(TypedCollection).name`  
Read / write name of typed tuple.

### Methods

`TypedFrozenset.copy()`

`TypedFrozenset.difference(expr)`

`TypedFrozenset.intersection(expr)`

`TypedFrozenset.isdisjoint(expr)`

`TypedFrozenset.issubset(expr)`

`TypedFrozenset.issuperset(expr)`

`(TypedCollection).new(tokens=None, item_class=None, name=None)`

`TypedFrozenset.symmetric_difference(expr)`

`TypedFrozenset.union(expr)`

### Special methods

`TypedFrozenset.__and__(expr)`

`(TypedCollection).__contains__(token)`

`(TypedCollection).__eq__(expr)`

`TypedFrozenset.__ge__(expr)`

`TypedFrozenset.__gt__(expr)`

`TypedFrozenset.__hash__()`

`(TypedCollection).__iter__()`

`TypedFrozenset.__le__(expr)`

`(TypedCollection).__len__()`

`TypedFrozenset.__lt__(expr)`

TypedFrozenset. **\_\_ne\_\_** (*expr*)

TypedFrozenset. **\_\_or\_\_** (*expr*)

(AbjadObject). **\_\_repr\_\_** ()

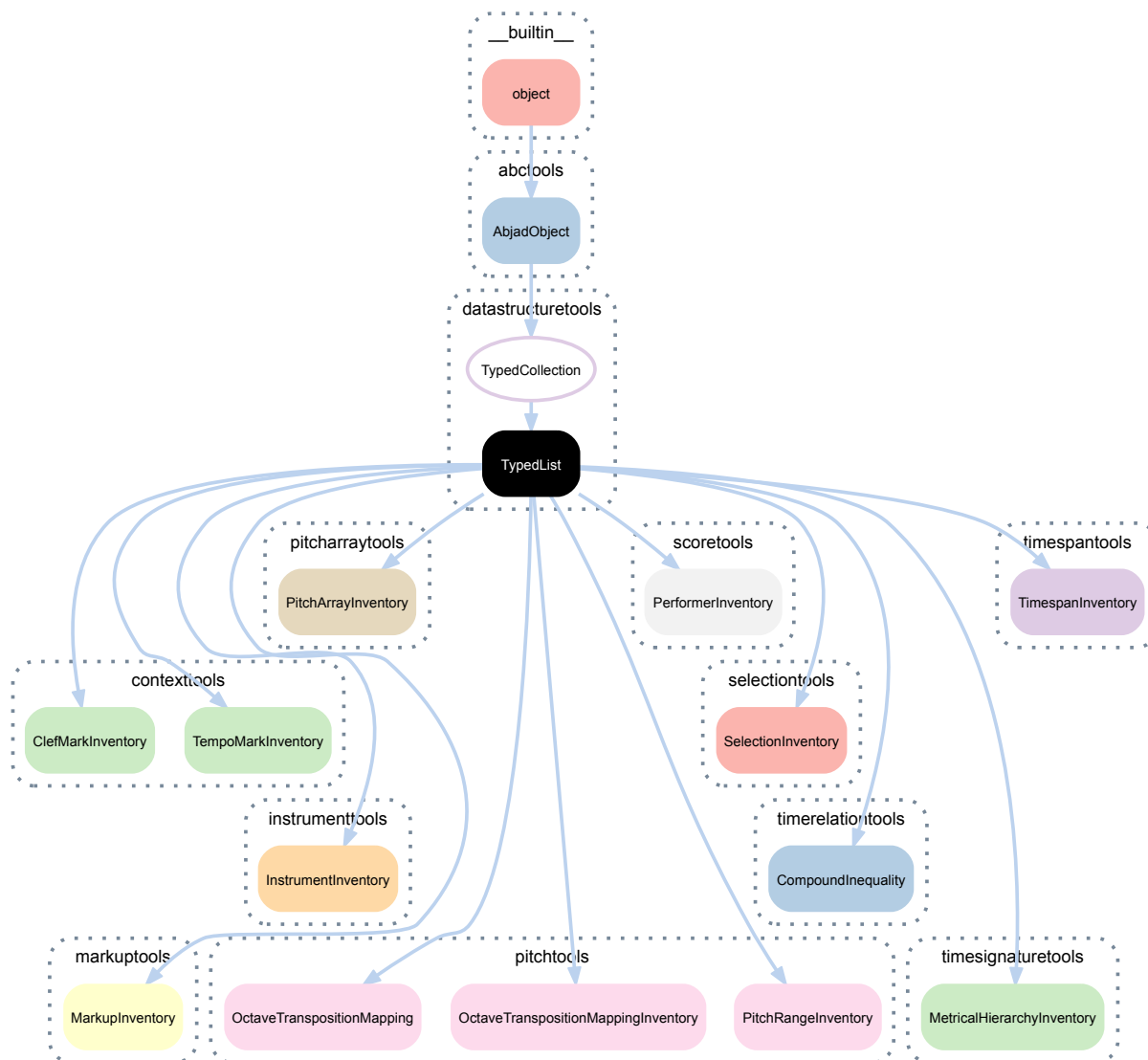
Interpreter representation of Abjad object.

Returns string.

TypedFrozenset. **\_\_sub\_\_** (*expr*)

TypedFrozenset. **\_\_xor\_\_** (*expr*)

## 51.2.14 datastructuretools.TypedList



**class** datastructuretools.**TypedList** (*tokens=None, item\_class=None, name=None*)

Ordered collection of objects, which optionally coerces its contents to the same type:

```
>>> object_collection = datastructuretools.TypedList()
>>> object_collection.append(23)
>>> object_collection.append('foo')
>>> object_collection.append(False)
>>> object_collection.append((1, 2, 3))
>>> object_collection.append(3.14159)
```

```
>>> print object_collection.storage_format
datastructuretools.TypedList([
    23,
    'foo',
    False,
    (1, 2, 3),
    3.14159
])
```

```
>>> pitch_collection = datastructuretools.TypedList(
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.append(0)
>>> pitch_collection.append("d' ")
>>> pitch_collection.append(('e', 4))
>>> pitch_collection.append(pitchtools.NamedPitch("f' "))
```

```
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
    pitchtools.NamedPitch("c' "),
    pitchtools.NamedPitch("d' "),
    pitchtools.NamedPitch("e' "),
    pitchtools.NamedPitch("f' ")
],
    item_class=pitchtools.NamedPitch
)
```

Implements the list interface.

## Bases

- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(TypedCollection).**item\_class**  
Item class to coerce tokens into.

(TypedCollection).**storage\_format**  
Storage format of typed tuple.

## Read/write properties

(TypedCollection).**name**  
Read / write name of typed tuple.

## Methods

TypedList.**append**(*token*)  
Change *token* to item and append:

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection[:]
[]
```

```
>>> integer_collection.append('1')
>>> integer_collection.append(2)
>>> integer_collection.append(3.4)
```

```
>>> integer_collection[:]
[1, 2, 3]
```

Returns none.

`TypedList.count(token)`

Change *token* to item and return count.

```
>>> integer_collection = datastructuretools.TypedList(
...     tokens=[0, 0., '0', 99],
...     item_class=int)
>>> integer_collection[:]
[0, 0, 0, 99]
```

```
>>> integer_collection.count(0)
3
```

Returns count.

`TypedList.extend(tokens)`

Change *tokens* to items and extend.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['0', 1.0, 2, 3.14159])
>>> integer_collection[:]
[0, 1, 2, 3]
```

Returns none.

`TypedList.index(token)`

Change *token* to item and return index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     tokens=('c'f', "as'", 'b', 'dss'),
...     item_class=pitchtools.NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

`TypedList.insert(i, token)`

Change *token* to item and insert.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['1', 2, 4.3])
>>> integer_collection[:]
[1, 2, 4]
```

```
>>> integer_collection.insert(0, '0')
>>> integer_collection[:]
[0, 1, 2, 4]
```

```
>>> integer_collection.insert(1, '9')
>>> integer_collection[:]
[0, 9, 1, 2, 4]
```

Returns none.

(TypedCollection).**new**(tokens=None, item\_class=None, name=None)

`TypedList.pop(i=-1)`

Aliases `list.pop()`.

`TypedList.remove(token)`

Change *token* to item and remove.

```
>>> integer_collection = datastructuretools.TypedList(
...     item_class=int)
>>> integer_collection.extend(['0', 1.0, 2, 3.14159])
>>> integer_collection[:]
[0, 1, 2, 3]
```

```
>>> integer_collection.remove('1')
>>> integer_collection[:]
[0, 2, 3]
```

Returns none.

`TypedList.reverse()`  
 Aliases `list.reverse()`.

`TypedList.sort(cmp=None, key=None, reverse=False)`  
 Aliases `list.sort()`.

## Special methods

`(TypedCollection).__contains__(token)`

`TypedList.__delitem__(i)`  
 Aliases `list.__delitem__()`.

`(TypedCollection).__eq__(expr)`

`TypedList.__getitem__(i)`  
 Aliases `list.__getitem__()`.

`TypedList.__iadd__(expr)`  
 Change tokens in *expr* to items and extend:

```
>>> dynamic_collection = datastructuretools.TypedList(
...     item_class=contexttools.DynamicMark)
>>> dynamic_collection.append('ppp')
>>> dynamic_collection += ['p', 'mp', 'mf', 'fff']
```

```
>>> print dynamic_collection.storage_format
datastructuretools.TypedList([
    contexttools.DynamicMark(
        'ppp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'p',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mp',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'mf',
        target_context=stafftools.Staff
    ),
    contexttools.DynamicMark(
        'fff',
        target_context=stafftools.Staff
    )
],
    item_class=contexttools.DynamicMark
)
```

Returns collection.

`(TypedCollection).__iter__()`

`(TypedCollection).__len__()`

(TypedCollection).**\_\_ne\_\_**(*expr*)

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

TypedList.**\_\_reversed\_\_**()

Aliases list.**\_\_reversed\_\_**().

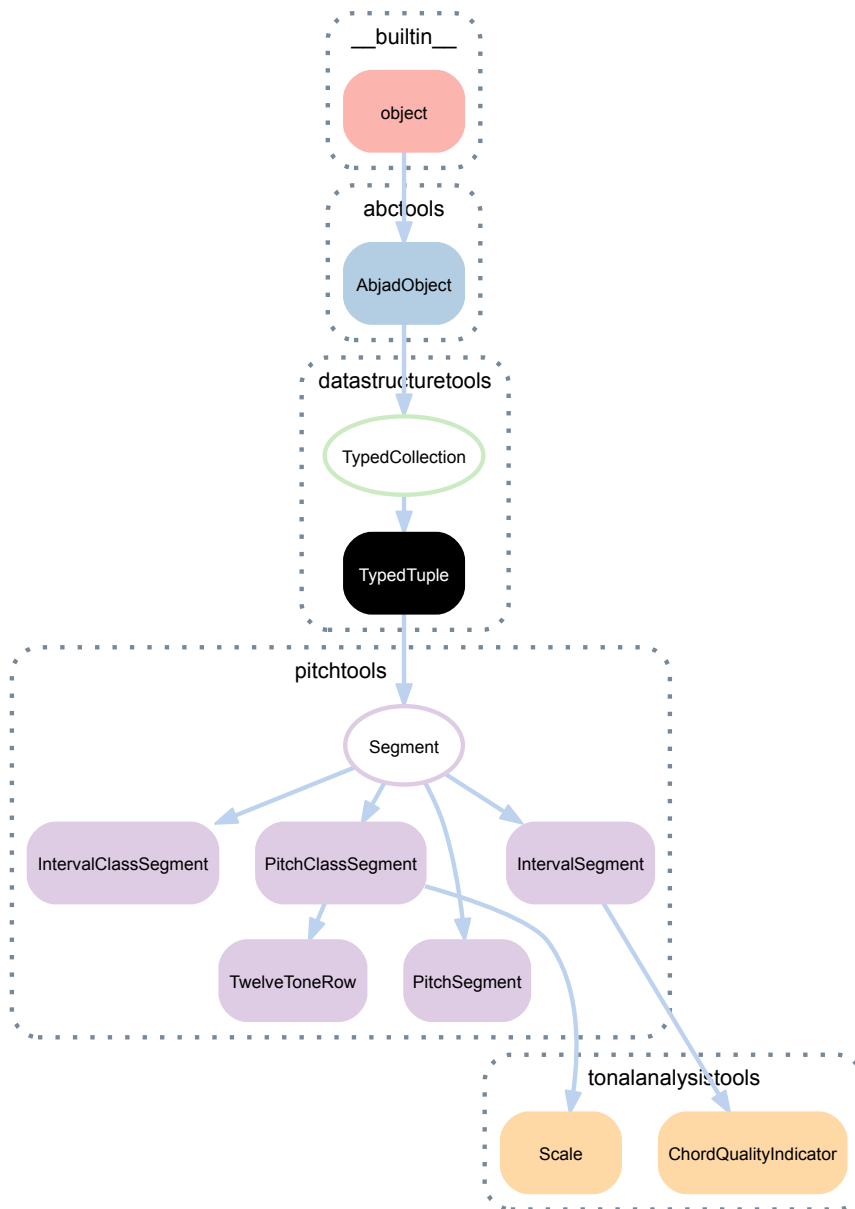
TypedList.**\_\_setitem\_\_**(*i*, *expr*)

Change tokens in *expr* to items and set:

```
>>> pitch_collection[-1] = 'gqs,'
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch('gqs,')
],
  item_class=pitchtools.NamedPitch
)
```

```
>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print pitch_collection.storage_format
datastructuretools.TypedList([
  pitchtools.NamedPitch("c'"),
  pitchtools.NamedPitch("d'"),
  pitchtools.NamedPitch("e'"),
  pitchtools.NamedPitch("f'"),
  pitchtools.NamedPitch("g'"),
  pitchtools.NamedPitch("a'"),
  pitchtools.NamedPitch("b'"),
  pitchtools.NamedPitch("c'")
],
  item_class=pitchtools.NamedPitch
)
```

### 51.2.15 datastructuretools.TypedTuple



**class** datastructuretools.**TypedTuple** (*tokens=None, item\_class=None, name=None*)

#### Bases

- datastructuretools.TypedCollection
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

(TypedCollection).**item\_class**  
Item class to coerce tokens into.

(TypedCollection).**storage\_format**  
Storage format of typed tuple.

## Read/write properties

(TypedCollection) **.name**  
Read / write name of typed tuple.

## Methods

TypedTuple **.count** (*token*)  
Change *token* to item and return count in collection.

TypedTuple **.index** (*token*)  
Change *token* to item and return index in collection.

(TypedCollection) **.new** (*tokens=None, item\_class=None, name=None*)

## Special methods

TypedTuple **.\_\_add\_\_** (*expr*)

TypedTuple **.\_\_contains\_\_** (*token*)  
Change *token* to item and return true if item exists in collection.

(TypedCollection) **.\_\_eq\_\_** (*expr*)

TypedTuple **.\_\_getitem\_\_** (*i*)  
Aliases tuple.\_\_getitem\_\_().

TypedTuple **.\_\_getslice\_\_** (*start, stop*)

TypedTuple **.\_\_hash\_\_** ()

(TypedCollection) **.\_\_iter\_\_** ()

(TypedCollection) **.\_\_len\_\_** ()

TypedTuple **.\_\_mul\_\_** (*expr*)

(TypedCollection) **.\_\_ne\_\_** (*expr*)

(AbjadObject) **.\_\_repr\_\_** ()  
Interpreter representation of Abjad object.  
Returns string.

TypedTuple **.\_\_rmul\_\_** (*expr*)



# DECORATORTOOLS

## 52.1 Functions

### 52.1.1 `decoratortools.requires`

`decoratortools.requires` (\*tests)  
Function decorator to require input parameter *tests*.

**Example:**

```
>>> @decoratortools.requires(  
...     mathtools.is_nonnegative_integer, string)  
>>> def multiply_string(n, string): return n * string
```

```
>>> multiply_string(2, 'bar')  
'barbar'
```

```
>>> multiply_string(2.5, 'bar')  
...  
AssertionError: is_nonnegative_integer(2.5) does not return true.
```

Decorator target is available like this:

```
>>> multiply_string.func_closure[1].cell_contents  
<function multiply_string at 0x104e512a8>
```

Decorator tests are available like this:

```
>>> multiply_string.func_closure[0].cell_contents  
(<function is_nonnegative_integer at 0x104725d70>, <type 'str'>)
```

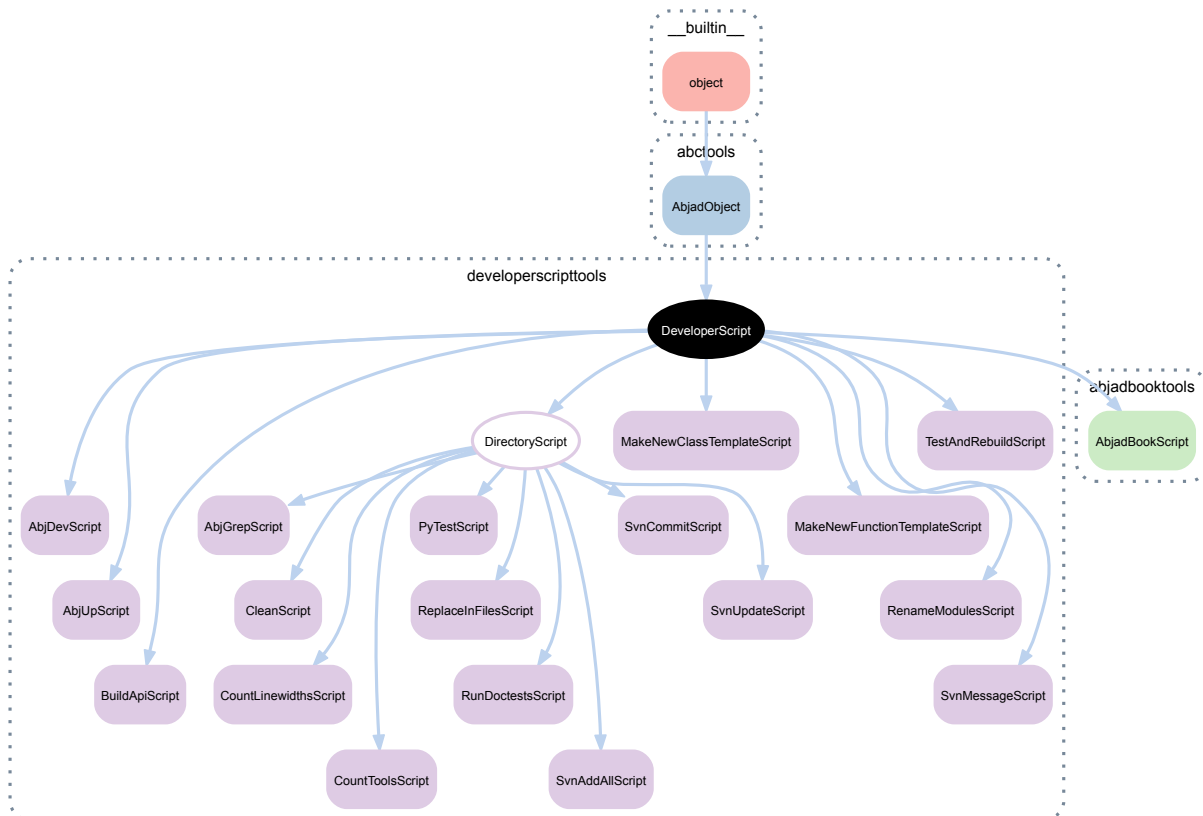
Returns decorated function in the form of function wrapper.



# DEVELOPERSCRIPTTOOLS

## 53.1 Abstract classes

### 53.1.1 developerscripttools.DeveloperScript



**class** `developerscripttools.DeveloperScript`

Abjad object-oriented model of a developer script.

*DeveloperScript* is the abstract parent from which concrete developer scripts inherit.

Developer scripts can be called from the command line, generally via the *ajv* command.

Developer scripts can be instantiated by other developer scripts in order to share functionality.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`DeveloperScript.alias`

The alias to use for the script, useful only if the script defines an abj-dev scripting group as well.

`DeveloperScript.argument_parser`

The script's instance of `argparse.ArgumentParser`.

`DeveloperScript.formatted_help`

`DeveloperScript.formatted_usage`

`DeveloperScript.formatted_version`

`DeveloperScript.long_description`

The long description, printed after arguments explanations.

`DeveloperScript.program_name`

The name of the script, callable from the command line.

`DeveloperScript.scripting_group`

The script's scripting subcommand group.

`DeveloperScript.short_description`

The short description of the script, printed before arguments explanations.

Also used as a summary in other contexts.

`DeveloperScript.version`

The version number of the script.

## Methods

`DeveloperScript.process_args(args)`

`DeveloperScript.setup_argument_parser()`

## Special methods

`DeveloperScript.__call__(args=None)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

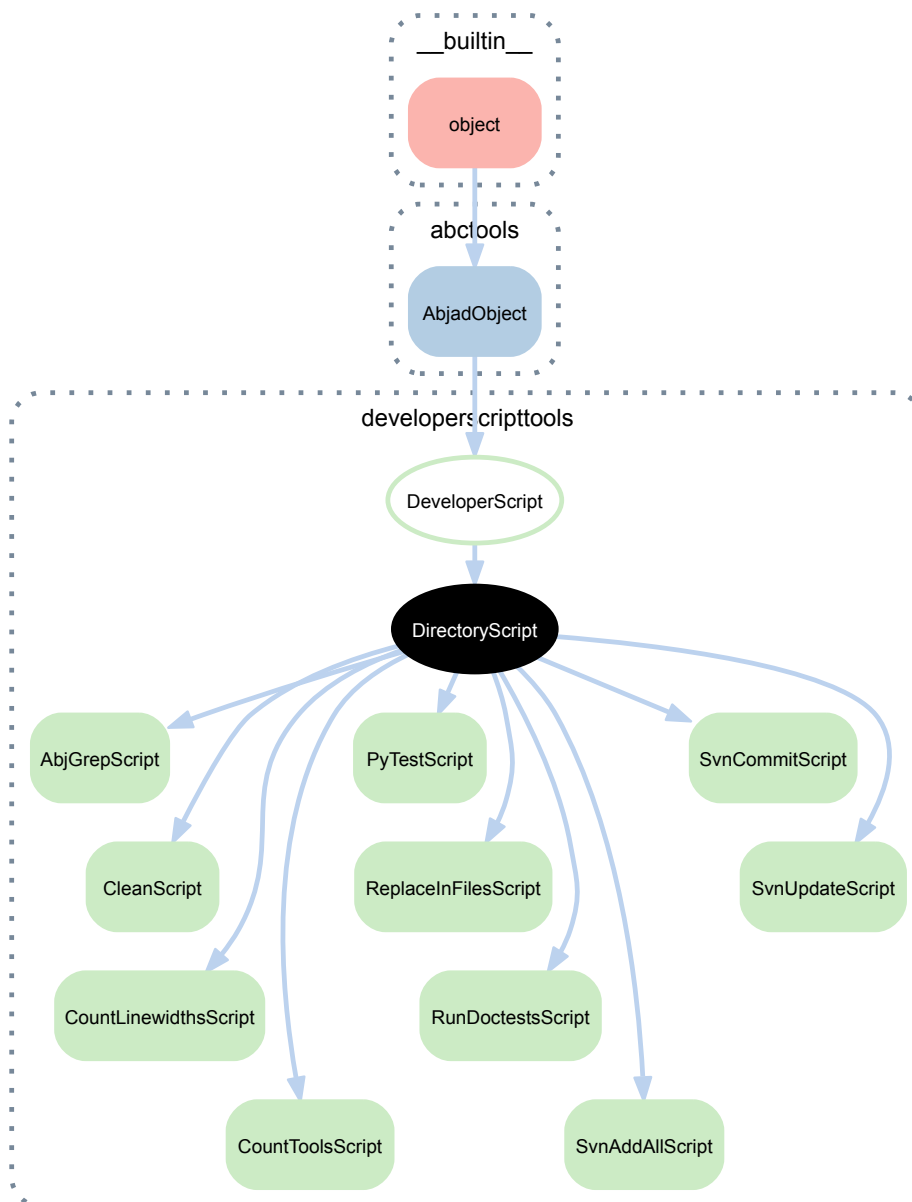
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 53.1.2 developerscripttools.DirectoryScript



**class** `developerscripttools.DirectoryScript`

*DirectoryScript* provides utilities for validating file system paths.

*DirectoryScript* is abstract.

#### Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(DeveloperScript).alias`

The alias to use for the script, useful only if the script defines an abj-dev scripting group as well.

`(DeveloperScript).argument_parser`  
The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`(DeveloperScript).long_description`  
The long description, printed after arguments explanations.

`(DeveloperScript).program_name`  
The name of the script, callable from the command line.

`(DeveloperScript).scripting_group`  
The script's scripting subcommand group.

`(DeveloperScript).short_description`  
The short description of the script, printed before arguments explanations.  
Also used as a summary in other contexts.

`(DeveloperScript).version`  
The version number of the script.

## Methods

`(DeveloperScript).process_args(args)`

`(DeveloperScript).setup_argument_parser()`

## Special methods

`(DeveloperScript).__call__(args=None)`

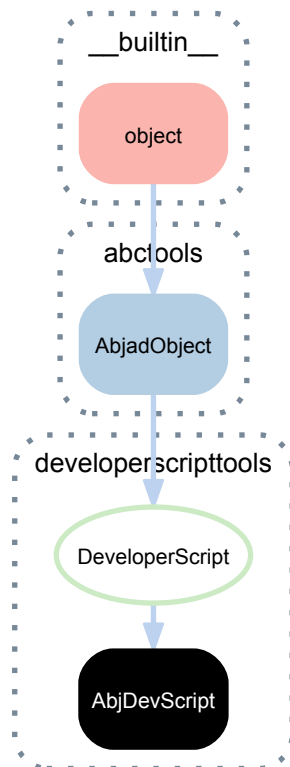
`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

## 53.2 Concrete classes

### 53.2.1 developerscripttools.AbjDevScript



**class** `developerscripttools.AbjDevScript`

*AbjDevScript* is the commandline entry-point to the Abjad developer scripts catalog.

Can be accessed on the commandline via *abj-dev* or *ajv*:

```

abjad$ ajv --help
usage: abj-dev [-h] [--version]

                {help,list,api,book,clean,count,dotest,grep,new,re,replace,svn,test,up}
                ...

Entry-point to Abjad developer scripts catalog.

optional arguments:
  -h, --help            show this help message and exit
  --version              show program's version number and exit

subcommands:
  {help,list,api,book,clean,count,dotest,grep,new,re,replace,svn,test,up}
  help                  print subcommand help
  list                  list subcommands
  api                   Build the Abjad APIs.
  book                  Preprocess HTML, LaTeX or ReST source with Abjad.
  clean                 Clean *.pyc, *.swp, __pycache__ and tmp* files and
                        folders from PATH.
  count                 "count"-related subcommands
  dotest                Run doctests on all modules in current path.
  grep                  grep PATTERN in PATH
  new                   "new"-related subcommands
  re                    Run py.test -x, dotest -x and then rebuild the API.
  rename                Rename public modules.
  replace               "replace"-related subcommands
  svn                   "svn"-related subcommands
  test                  Run "py.test" on various Abjad paths.
  up                    run `ajv svn up -R -C`
  
```

*ajv* supports subcommands similar to *svn*:

```
abjad$ ajv api --help
usage: build-api [-h] [--version] [-M] [-X] [-C] [-O] [--format FORMAT]

Build the Abjad APIs.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -M, --mainline        build the mainline API
  -X, --experimental    build the experimental API
  -C, --clean           run "make clean" before building the api
  -O, --open            open the docs in a web browser after building
  --format FORMAT       Sphinx builder to use
```

Return *AbjDevScript* instance.

## Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(DeveloperScript).alias`

The alias to use for the script, useful only if the script defines an abj-dev scripting group as well.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`AbjDevScript.developer_script_aliases`

`AbjDevScript.developer_script_classes`

`AbjDevScript.developer_script_program_names`

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`AbjDevScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`(DeveloperScript).scripting_group`

The script's scripting subcommand group.

`AbjDevScript.short_description`

`AbjDevScript.version`

## Methods

`AbjDevScript.process_args(args)`

`AbjDevScript.setup_argument_parser(parser)`



## Special methods

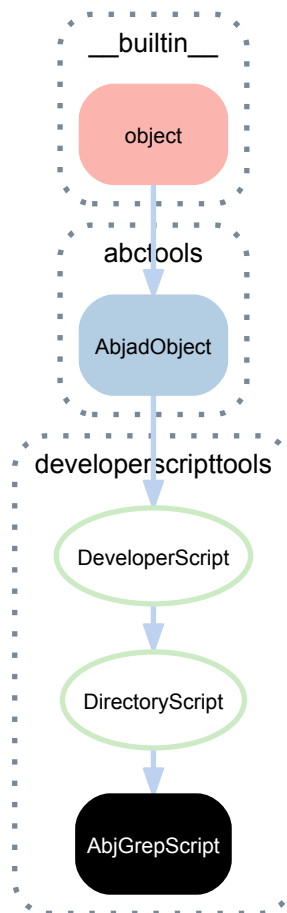
`AbjadDevScript.__call__(args=None)`

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

## 53.2.2 developerscripttools.AbjGrepScript



**class** `developerscripttools.AbjGrepScript`

Run *grep* against a path, ignoring *svn* and docs-related files:

```

abjad$ ajv grep --help
usage: abj-grep [-h] [--version] [-W] [-P PATH | -X | -M | -T | -R] pattern

grep PATTERN in PATH

positional arguments:
  pattern                pattern to search for

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -W, --whole-line      match whole lines only
  -P PATH, -X           search in PATH
  -M, --multiline       match across multiple lines
  -T, --text            treat the input as plain text
  -R, --recursive       recursively search in all directories under PATH
  
```

```
-h, --help            show this help message and exit
--version             show program's version number and exit
-W, --whole-words-only match only whole words, similar to grep's "-w" flag
-P PATH, --path PATH  grep PATH
-X, --experimental    grep Abjad abjad.tools directory
-M, --mainline        grep Abjad mainline directory
-T, --tools           grep Abjad mainline tools directory
-R, --root            grep Abjad root directory
```

If no PATH flag is specified, the current directory will be searched.

Return *AbjGrepScript* instance.

## Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`AbjGrepScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`AbjGrepScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`AbjGrepScript.scripting_group`

`AbjGrepScript.short_description`

`AbjGrepScript.version`

## Methods

`AbjGrepScript.process_args(args)`

`AbjGrepScript.setup_argument_parser(parser)`

## Special methods

`(DeveloperScript).__call__(args=None)`

`(AbjadObject).__eq__(expr)`

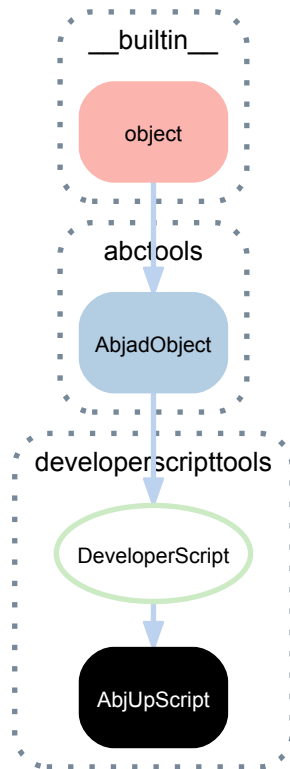
True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 53.2.3 developerscripttools.AbjUpScript



**class** developerscripttools.**AbjUpScript**

Run *ajv svn up -R -C*:

```
abjad$ ajv up --help
usage: abj-up [-h] [--version]

run `ajv svn up -R -C`

optional arguments:
  -h, --help  show this help message and exit
  --version  show program's version number and exit
```

Return *AbjUpScript* instance.

#### Bases

- developerscripttools.DeveloperScript
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

`AbjUpScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`AbjUpScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`AbjUpScript.scripting_group`

`AbjUpScript.short_description`

`AbjUpScript.version`

## Methods

`AbjUpScript.process_args(args)`

`AbjUpScript.setup_argument_parser(parser)`

## Special methods

`(DeveloperScript).__call__(args=None)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

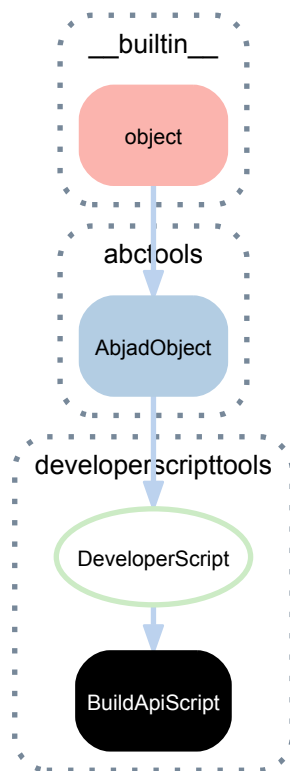
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 53.2.4 developerscripttools.BuildApiScript



**class** `developerscripttools.BuildApiScript`

Build the Abjad APIs:

```

abjad$ ajv api --help
usage: build-api [-h] [--version] [-M] [-X] [-C] [-O] [--format FORMAT]

Build the Abjad APIs.

optional arguments:
  -h, --help            show this help message and exit
  --version              show program's version number and exit
  -M, --mainline         build the mainline API
  -X, --experimental     build the experimental API
  -C, --clean            run "make clean" before building the api
  -O, --open             open the docs in a web browser after building
  --format FORMAT        Sphinx builder to use
  
```

Return *BuildApiScript* instance.

#### Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`BuildApiScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`  
`(DeveloperScript).formatted_version`  
`BuildApiScript.long_description`  
`(DeveloperScript).program_name`  
    The name of the script, callable from the command line.  
`BuildApiScript.scripting_group`  
`BuildApiScript.short_description`  
`BuildApiScript.version`

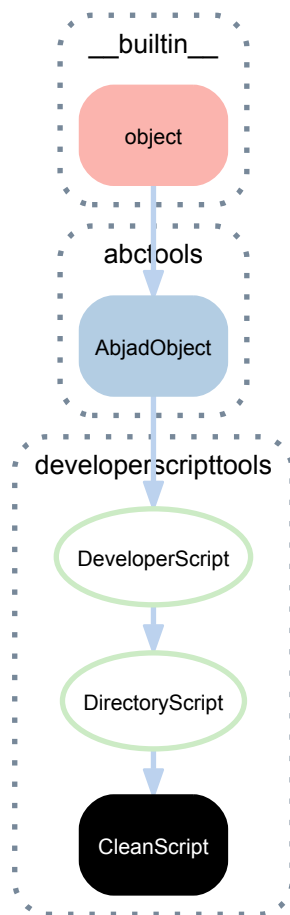
## Methods

`BuildApiScript.process_args` (*args*)  
`BuildApiScript.setup_argument_parser` (*parser*)

## Special methods

`(DeveloperScript).__call__` (*args=None*)  
`(AbjadObject).__eq__` (*expr*)  
    True when ID of *expr* equals ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__ne__` (*expr*)  
    True when ID of *expr* does not equal ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__repr__` ()  
    Interpreter representation of Abjad object.  
    Returns string.

### 53.2.5 developerscripttools.CleanScript



**class** `developerscripttools.CleanScript`

Remove `.pyc`, `*.swp` files and `__pycache__` and `tmp` directories recursively in a path:

```

abjad$ ajv clean --help
usage: clean [-h] [--version] [--pyc] [--pycache] [--swp] [--tmp] [path]

Clean *.pyc, *.swp, __pycache__ and tmp* files and folders from PATH.

positional arguments:
  path                directory tree to be recursed over

optional arguments:
  -h, --help          show this help message and exit
  --version            show program's version number and exit
  --pyc               delete *.pyc files
  --pycache            delete __pycache__ folders
  --swp               delete Vim *.swp file
  --tmp               delete tmp* folders
  
```

Return *CleanScript* instance.

#### Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`CleanScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`CleanScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`CleanScript.scripting_group`

`CleanScript.short_description`

`CleanScript.version`

## Methods

`CleanScript.process_args(args)`

`CleanScript.setup_argument_parser(parser)`

## Special methods

`(DeveloperScript).__call__(args=None)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

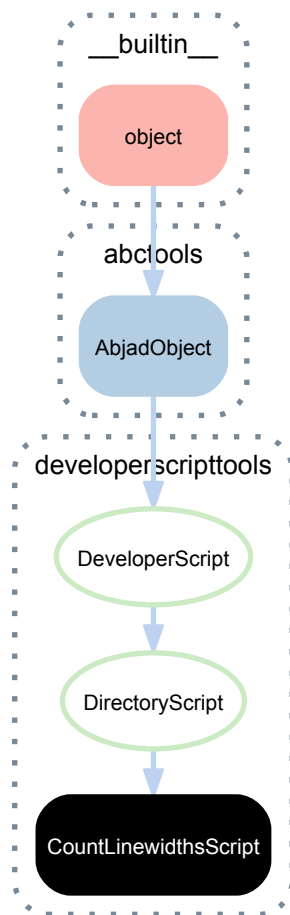
`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.



### 53.2.6 developerscripttools.CountLinewidthsScript



**class** `developerscripttools.CountLinewidthsScript`

Tabulate the linewidths of modules in a path:

```

abjad$ ajv count linewidths --help
usage: count-linewidths [-h] [--version] [-l N] [-o w|m] [-C | -D] [-a | -d]
                        [-gt N | -lt N | -eq N]
                        path

Count maximum line-width of all modules in PATH.

positional arguments:
  path                directory tree to be recursed over

optional arguments:
  -h, --help            show this help message and exit
  --version            show program's version number and exit
  -l N, --limit N      limit output to last N items
  -o w|m, --order-by w|m
                        order by line width [w] or module name [m]
  -C, --code           count linewidths of all code in module
  -D, --docstrings     count linewidths of all docstrings in module
  -a, --ascending      sort results ascending
  -d, --descending     sort results descending
  -gt N, --greater-than N
                        line widths greater than N
  -lt N, --less-than N
                        line widths less than N
  -eq N, --equal-to N
                        line widths equal to N
  
```

Return *CountLinewidthsScript* instance.

## Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`CountLinewidthsScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`CountLinewidthsScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`CountLinewidthsScript.scripting_group`

`CountLinewidthsScript.short_description`

`CountLinewidthsScript.version`

## Methods

`CountLinewidthsScript.process_args(args)`

`CountLinewidthsScript.setup_argument_parser(parser)`

## Special methods

`(DeveloperScript).__call__(args=None)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

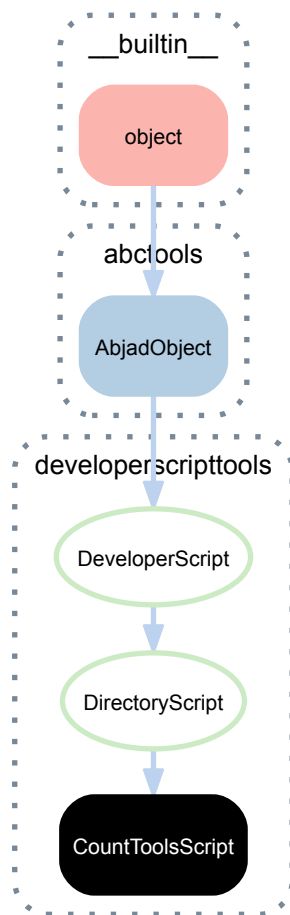
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 53.2.7 developerscripttools.CountToolsScript



**class** `developerscripttools.CountToolsScript`  
 Count public and private functions and classes in a path:

```

abjad$ ajv count tools --help
usage: count-tools [-h] [--version] [-v] path

Count tools in PATH.

positional arguments:
  path                directory tree to be recursed over

optional arguments:
  -h, --help          show this help message and exit
  --version            show program's version number and exit
  -v, --verbose        print verbose information
  
```

Return *CountToolsScript* instance.

#### Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`CountToolsScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`CountToolsScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`CountToolsScript.scripting_group`

`CountToolsScript.short_description`

`CountToolsScript.version`

## Methods

`CountToolsScript.process_args(args)`

`CountToolsScript.setup_argument_parser(parser)`

## Special methods

`(DeveloperScript).__call__(args=None)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

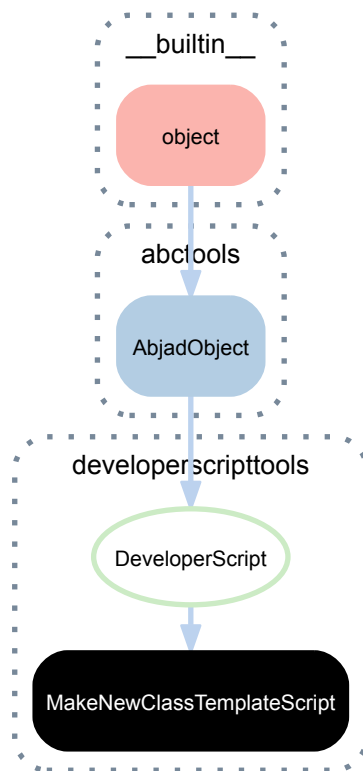
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 53.2.8 developerscripttools.MakeNewClassTemplateScript



**class** `developerscripttools.MakeNewClassTemplateScript`

Create class stubs, complete with test subdirectory:

```

abjad$ ajv new class --help
usage: make-new-class-template [-h] [--version] (-X | -M) name

Make a new class template file.

positional arguments:
  name                tools package qualified class name

optional arguments:
  -h, --help          show this help message and exit
  --version            show program's version number and exit
  -X, --experimental  use the Abjad experimental tools path
  -M, --mainline      use the Abjad mainline tools path
  
```

Return *MakeNewClassTemplateScript* instance.

#### Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`MakeNewClassTemplateScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`  
`(DeveloperScript).formatted_version`  
`MakeNewClassTemplateScript.long_description`  
`(DeveloperScript).program_name`  
    The name of the script, callable from the command line.  
`MakeNewClassTemplateScript.scripting_group`  
`MakeNewClassTemplateScript.short_description`  
`MakeNewClassTemplateScript.version`

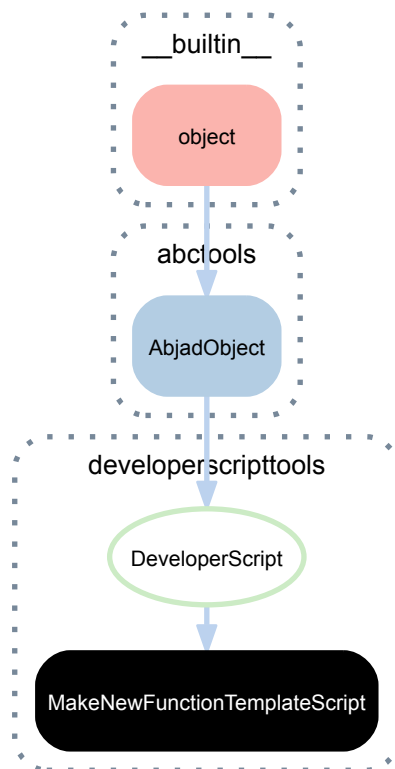
## Methods

`MakeNewClassTemplateScript.process_args` (*args*)  
`MakeNewClassTemplateScript.setup_argument_parser` (*parser*)

## Special methods

`(DeveloperScript).__call__` (*args=None*)  
`(AbjadObject).__eq__` (*expr*)  
    True when ID of *expr* equals ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__ne__` (*expr*)  
    True when ID of *expr* does not equal ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__repr__` ()  
    Interpreter representation of Abjad object.  
    Returns string.

### 53.2.9 developerscripttools.MakeNewFunctionTemplateScript



**class** `developerscripttools.MakeNewFunctionTemplateScript`

Create function stub files:

```

abjad$ ajv new function --help
usage: make-new-function-template [-h] [--version] (-X | -M) name

Make a new function template file.

positional arguments:
  name                tools package qualified function name

optional arguments:
  -h, --help          show this help message and exit
  --version            show program's version number and exit
  -X, --experimental  use the Abjad experimental tools path
  -M, --mainline      use the Abjad mainline tools path
  
```

Return *MakeNewFunctionTemplateScript* instance.

#### Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`MakeNewFunctionTemplateScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`  
`(DeveloperScript).formatted_version`  
`MakeNewFunctionTemplateScript.long_description`  
`(DeveloperScript).program_name`  
    The name of the script, callable from the command line.  
`MakeNewFunctionTemplateScript.scripting_group`  
`MakeNewFunctionTemplateScript.short_description`  
`MakeNewFunctionTemplateScript.version`

## Methods

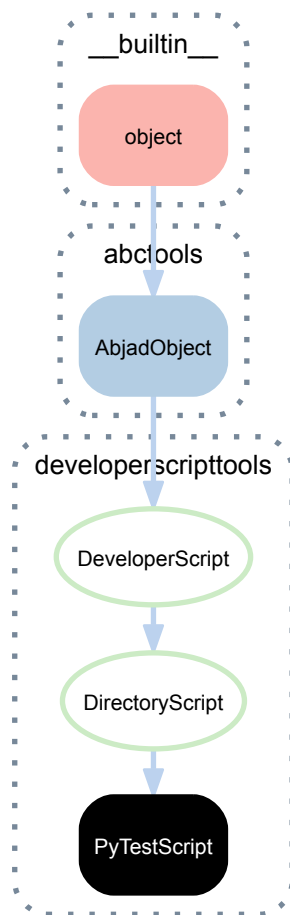
`MakeNewFunctionTemplateScript.process_args` (*args*)  
`MakeNewFunctionTemplateScript.setup_argument_parser` (*parser*)

## Special methods

`(DeveloperScript).__call__` (*args=None*)  
`(AbjadObject).__eq__` (*expr*)  
    True when ID of *expr* equals ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__ne__` (*expr*)  
    True when ID of *expr* does not equal ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__repr__` ()  
    Interpreter representation of Abjad object.  
    Returns string.



### 53.2.10 developerscripttools.PyTestScript



**class** `developerscripttools.PyTestScript`

Run *py.test* on various Abjad paths:

```

abjad$ ajv test --help
usage: py-test [-h] [--version] [-p] [-r chars] [-x] [-A | -D | -M | -X]

Run "py.test" on various Abjad paths.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -p, --parallel        run py.test with multiprocessing
  -r chars, --report chars
                        show extra test summary info as specified by chars
                        (f)ailed, (E)rror, (s)kipped, (x)failed, (X)passed.
  -x, --exitfirst       stop on first failure
  -A, --all             test all directories, including demos
  -D, --demos           test demos directory
  -M, --mainline        test mainline tools directory
  -X, --experimental    test experimental directory
  
```

Return *PyTestScript* instance.

#### Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`PyTestScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`PyTestScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`PyTestScript.scripting_group`

`PyTestScript.short_description`

`PyTestScript.version`

## Methods

`PyTestScript.process_args(args)`

`PyTestScript.setup_argument_parser(parser)`

## Special methods

`(DeveloperScript).__call__(args=None)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

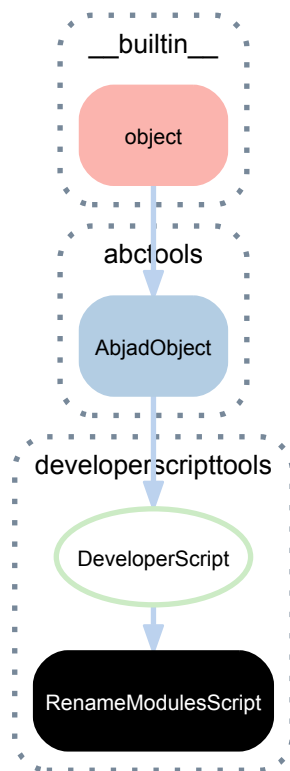
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 53.2.11 developerscripttools.RenameModulesScript



**class** `developerscripttools.RenameModulesScript`

Rename classes and functions.

Handle renaming the module and package, as well as any tests, documentation or mentions of the class throughout the Abjad codebase:

```

abjad$ ajv rename --help
usage: rename-modules [-h] [--version] (-C | -F)

Rename public modules.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -C, --classes         rename classes
  -F, --functions       rename functions
  
```

Return *RenameModulesScript* instance.

#### Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`RenameModulesScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`  
`(DeveloperScript).formatted_version`  
`RenameModulesScript.long_description`  
`(DeveloperScript).program_name`  
    The name of the script, callable from the command line.  
`RenameModulesScript.scripting_group`  
`RenameModulesScript.short_description`  
`RenameModulesScript.version`

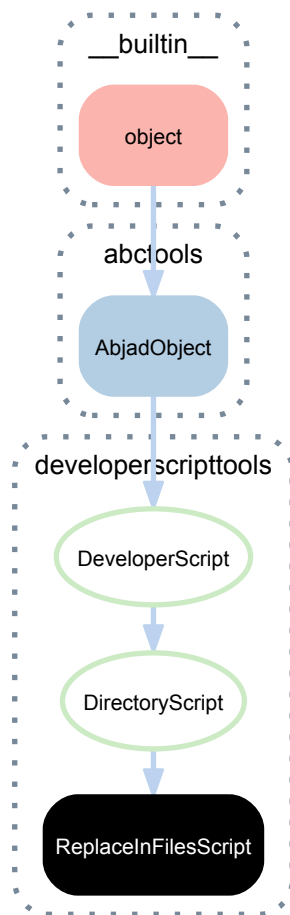
## Methods

`RenameModulesScript.process_args` (*args*)  
`RenameModulesScript.setup_argument_parser` (*parser*)

## Special methods

`(DeveloperScript).__call__` (*args=None*)  
`(AbjadObject).__eq__` (*expr*)  
    True when ID of *expr* equals ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__ne__` (*expr*)  
    True when ID of *expr* does not equal ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__repr__` ()  
    Interpreter representation of Abjad object.  
    Returns string.

### 53.2.12 developerscripttools.ReplaceInFilesScript



#### class developerscripttools.ReplaceInFilesScript

Replace text in files recursively:

```

abjad$ ajv replace text --help
usage: replace-in-files [-h] [--version] [--verbose] [-Y] [-R] [-W]
                        [-F PATTERN] [-D PATTERN]
                        [path] old new

Replace text.

positional arguments:
  path                directory tree to be recursed over
  old                 old text
  new                 new text

optional arguments:
  -h, --help          show this help message and exit
  --version            show program's version number and exit
  --verbose            print replacement info even when --force flag is set.
  -Y, --force          force "yes" to every replacement
  -R, --regex          treat "old" as a regular expression
  -W, --whole-words-only
                        match only whole words, similar to grep's "-w" flag
  -F PATTERN, --without-files PATTERN
                        Exclude files matching pattern(s)
  -D PATTERN, --without-dirs PATTERN
                        Exclude folders matching pattern(s)
  
```

Multiple patterns for excluding files or folders can be specified by restating the *--without-files* or *--without-dirs* commands:

```
abjad$ ajv replace text . foo bar -F *.txt -F *.rst -F *.htm
```

Return *ReplaceInFilesScript* instance.

## Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ReplaceInFilesScript.alias`

(`DeveloperScript`) `.argument_parser`

The script's instance of `argparse.ArgumentParser`.

(`DeveloperScript`) `.formatted_help`

(`DeveloperScript`) `.formatted_usage`

(`DeveloperScript`) `.formatted_version`

`ReplaceInFilesScript.long_description`

(`DeveloperScript`) `.program_name`

The name of the script, callable from the command line.

`ReplaceInFilesScript.scripting_group`

`ReplaceInFilesScript.short_description`

`ReplaceInFilesScript.skipped_directories`

`ReplaceInFilesScript.skipped_files`

`ReplaceInFilesScript.version`

## Methods

`ReplaceInFilesScript.process_args` (*args*)

`ReplaceInFilesScript.setup_argument_parser` (*parser*)

## Special methods

(`DeveloperScript`) `.__call__` (*args=None*)

(`AbjadObject`) `.__eq__` (*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(`AbjadObject`) `.__ne__` (*expr*)

True when ID of *expr* does not equal ID of Abjad object.

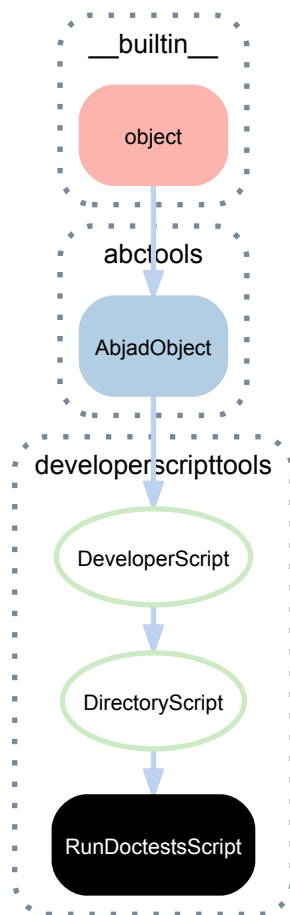
Returns boolean.

(`AbjadObject`) `.__repr__` ()

Interpreter representation of Abjad object.

Returns string.

### 53.2.13 developerscripttools.RunDoctestsScript



**class** `developerscripttools.RunDoctestsScript`

Run doctests on all Python files in current directory recursively:

```

abjad$ ajv doctest --help
usage: run-doctests [-h] [--version] [--diff]

Run doctests on all modules in current path.

optional arguments:
  -h, --help      show this help message and exit
  --version       show program's version number and exit
  --diff          print diff-like output on failed tests.
  
```

Return *RunDoctestsScript* instance.

#### Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`RunDoctestsScript.alias`

`(DeveloperScript).argument_parser`  
The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`RunDoctestsScript.long_description`

`(DeveloperScript).program_name`  
The name of the script, callable from the command line.

`RunDoctestsScript.scripting_group`

`RunDoctestsScript.short_description`

`RunDoctestsScript.version`

## Methods

`RunDoctestsScript.process_args` (*args*)

`RunDoctestsScript.setup_argument_parser` (*parser*)

## Special methods

`(DeveloperScript).__call__` (*args=None*)

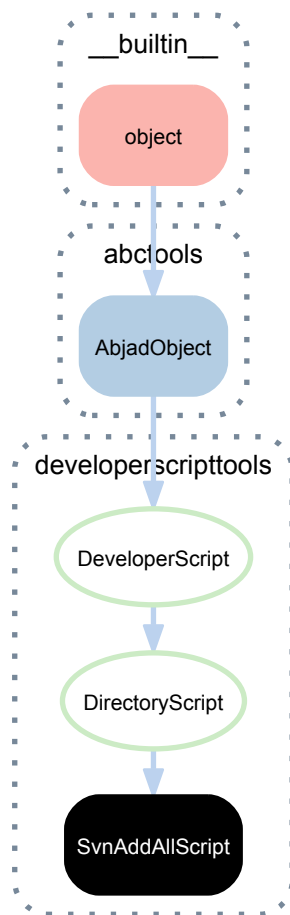
`(AbjadObject).__eq__` (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__` (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__` ()  
Interpreter representation of Abjad object.  
Returns string.



### 53.2.14 developerscripttools.SvnAddAllScript



**class** `developerscripttools.SvnAddAllScript`

Run *svn add* on all unversioned files in path:

```

abjad$ ajv svn add --help
usage: svn-add-all [-h] [--version] [path]

"svn add" all unversioned files in PATH.

positional arguments:
  path                directory tree to be recursed over

optional arguments:
  -h, --help          show this help message and exit
  --version            show program's version number and exit
  
```

Return *SvnAddAllScript* instance.

#### Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`SvnAddAllScript.alias`

`(DeveloperScript).argument_parser`  
The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`SvnAddAllScript.long_description`

`(DeveloperScript).program_name`  
The name of the script, callable from the command line.

`SvnAddAllScript.scripting_group`

`SvnAddAllScript.short_description`

`SvnAddAllScript.version`

## Methods

`SvnAddAllScript.process_args` (*args*)

`SvnAddAllScript.setup_argument_parser` (*parser*)

## Special methods

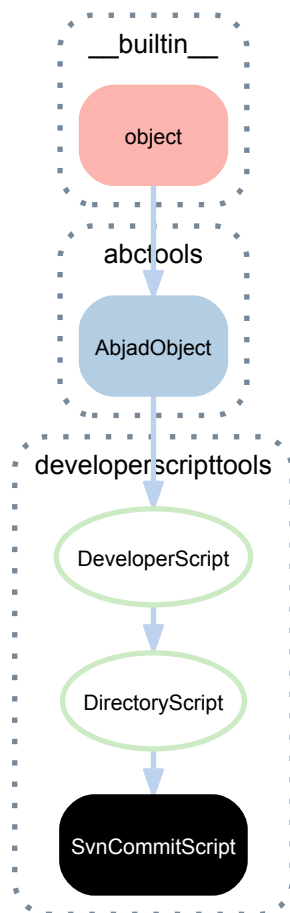
`(DeveloperScript).__call__` (*args=None*)

`(AbjadObject).__eq__` (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__` (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__` ()  
Interpreter representation of Abjad object.  
Returns string.

### 53.2.15 developerscripttools.SvnCommitScript



**class** `developerscripttools.SvnCommitScript`

Run *svn commit*, using the commit message stored in the *.abjad* directory.

The commit message will be printed to the terminal, and must be manually accepted or rejected before proceeding:

```

abjad$ ajv svn ci --help
usage: svn-commit [-h] [--version] [path]

"svn commit", using previously written commit message.

positional arguments:
  path                commit the path PATH

optional arguments:
  -h, --help          show this help message and exit
  --version            show program's version number and exit
  
```

Return *SvnCommitScript* instance.

#### Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`SvnCommitScript.alias`  
(DeveloperScript).`argument_parser`  
The script's instance of `argparse.ArgumentParser`.  
(DeveloperScript).`formatted_help`  
(DeveloperScript).`formatted_usage`  
(DeveloperScript).`formatted_version`  
`SvnCommitScript.long_description`  
(DeveloperScript).`program_name`  
The name of the script, callable from the command line.  
`SvnCommitScript.scripting_group`  
`SvnCommitScript.short_description`  
`SvnCommitScript.version`

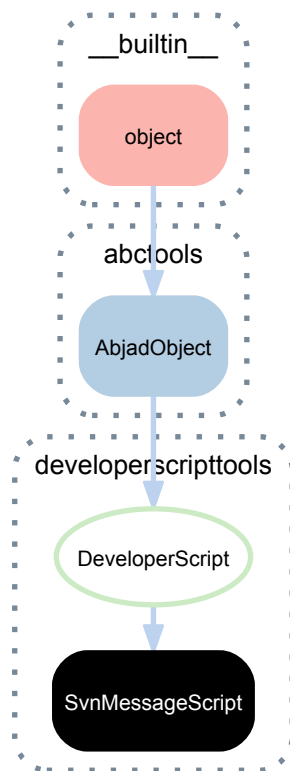
## Methods

`SvnCommitScript.process_args` (*args*)  
`SvnCommitScript.setup_argument_parser` (*parser*)

## Special methods

(DeveloperScript).`__call__` (*args=None*)  
(AbjadObject).`__eq__` (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.  
(AbjadObject).`__ne__` (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.  
(AbjadObject).`__repr__` ()  
Interpreter representation of Abjad object.  
Returns string.

### 53.2.16 developerscripttools.SvnMessageScript



**class** `developerscripttools.SvnMessageScript`

Edit a temporary *svn* commit message, stored in the *.abjad* directory:

```

abjad$ ajv svn msg --help
usage: svn-message [-h] [--version] [-C]

Write commit message for future commit usage.

optional arguments:
  -h, --help      show this help message and exit
  --version       show program's version number and exit
  -C, --clean     delete previous commit message before editing
  
```

Return *SvnMessageScript* instance.

#### Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`SvnMessageScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`SvnMessageScript.commit_message_path`

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`  
`SvnMessageScript.long_description`  
`(DeveloperScript).program_name`  
    The name of the script, callable from the command line.  
`SvnMessageScript.scripting_group`  
`SvnMessageScript.short_description`  
`SvnMessageScript.version`

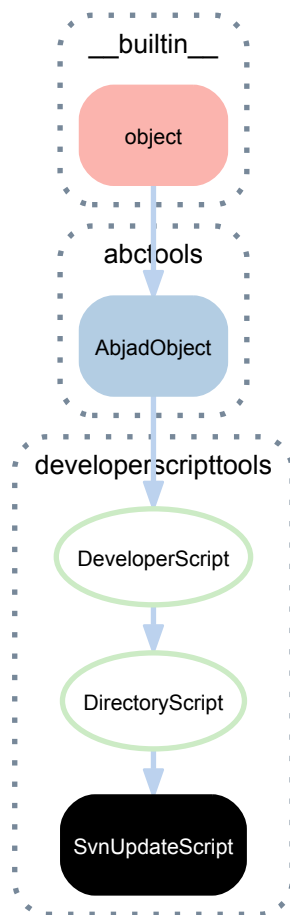
## Methods

`SvnMessageScript.process_args` (*args*)  
`SvnMessageScript.setup_argument_parser` (*parser*)

## Special methods

`(DeveloperScript).__call__` (*args=None*)  
`(AbjadObject).__eq__` (*expr*)  
    True when ID of *expr* equals ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__ne__` (*expr*)  
    True when ID of *expr* does not equal ID of Abjad object.  
    Returns boolean.  
`(AbjadObject).__repr__` ()  
    Interpreter representation of Abjad object.  
    Returns string.

### 53.2.17 developerscripttools.SvnUpdateScript



**class** `developerscripttools.SvnUpdateScript`

Run *svn up* on various Abjad paths:

```

abjad$ ajv svn up --help
usage: svn-update [-h] [--version] [-C] [-P PATH | -E | -M | -R]

"svn update" various paths.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -C, --clean           remove .pyc files and __pycache__ directories before
                        updating
  -P PATH, --path PATH  update the path PATH
  -E, --experimental    update Abjad abjad.tools directory
  -M, --mainline        update Abjad mainline directory
  -R, --root            update Abjad root directory

If no path flag is specified, the current directory will be updated.
  
```

It is usually most useful to run the script with the *-clean* flag, in case there are incoming deletes, as *svn* will not delete directories containing unversioned files, such as *.pyc*:

```
bash$ ajv svn up -C -R
```

Return *SvnUpdateScript* instance.

#### Bases

- `developerscripttools.DirectoryScript`

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`SvnUpdateScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`SvnUpdateScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`SvnUpdateScript.scripting_group`

`SvnUpdateScript.short_description`

`SvnUpdateScript.version`

## Methods

`SvnUpdateScript.process_args(args)`

`SvnUpdateScript.setup_argument_parser(parser)`

## Special methods

`(DeveloperScript).__call__(args=None)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

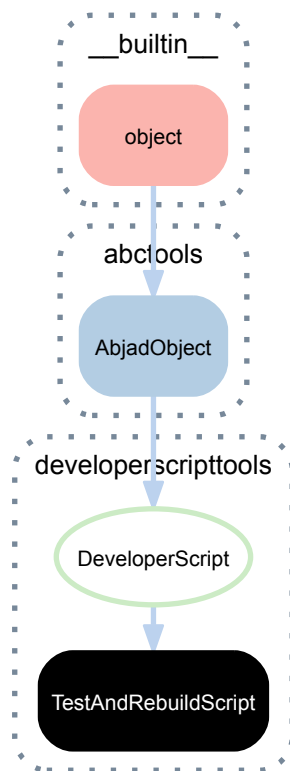
`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.



### 53.2.18 developerscripttools.TestAndRebuildScript



**class** `developerscripttools.TestAndRebuildScript`

#### Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`TestAndRebuildScript.alias`

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).formatted_help`

`(DeveloperScript).formatted_usage`

`(DeveloperScript).formatted_version`

`TestAndRebuildScript.long_description`

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`TestAndRebuildScript.scripting_group`

`TestAndRebuildScript.short_description`

`TestAndRebuildScript.version`

## Methods

`TestAndRebuildScript.get_terminal_width()`

Borrowed from the `py` lib.

`TestAndRebuildScript.process_args(args)`

`TestAndRebuildScript.rebuild_docs(args)`

`TestAndRebuildScript.run_doctest(args)`

`TestAndRebuildScript.run_pytest(args)`

`TestAndRebuildScript.setup_argument_parser(parser)`

## Special methods

`(DeveloperScript).__call__(args=None)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 53.3 Functions

### 53.3.1 `developerscripttools.get_developer_script_classes`

`developerscripttools.get_developer_script_classes()`

Returns a list of all developer script classes.

### 53.3.2 `developerscripttools.run_abjadbook`

`developerscripttools.run_abjadbook()`

Entry point for `setuptools`.

One-line wrapper around `AbjadBookScript`.

### 53.3.3 `developerscripttools.run_abjdev`

`developerscripttools.run_abjdev()`

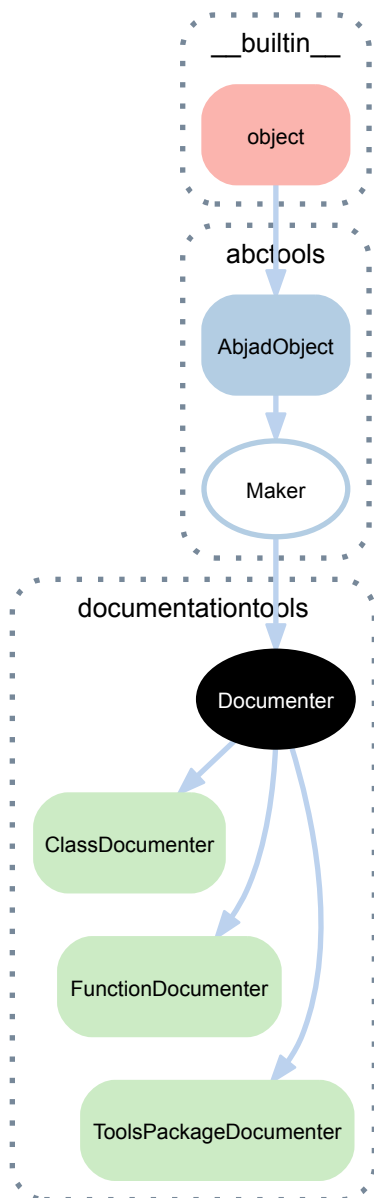
Entry point for `setuptools`.

One-line wrapper around `AbjDevScript`.

## DOCUMENTATIONTOOLS

### 54.1 Abstract classes

#### 54.1.1 documentationtools.Documenter



**class** `documentationtools.Documenter` (*obj*, *prefix*=*'abjad.tools.'*)  
Documenter is an abstract base class for documentation classes.

### Bases

- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`Documenter.module_name`

`Documenter.object`

`Documenter.prefix`

`(Maker).storage_format`  
Storage format of maker.

Returns string.

### Methods

`Documenter.new` (*obj*=*None*, *prefix*=*None*)

### Static methods

`Documenter.write` (*file\_path*, *restructured\_text*)

### Special methods

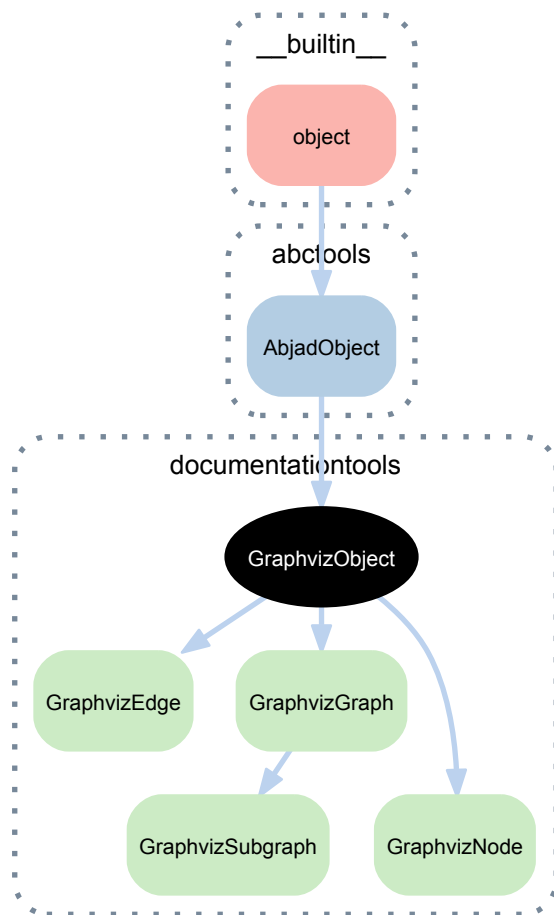
`(Maker).__call__()`

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

### 54.1.2 documentationtools.GraphvizObject



**class** `documentationtools.GraphvizObject` (*attributes=None*)  
 An attributed Graphviz object.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

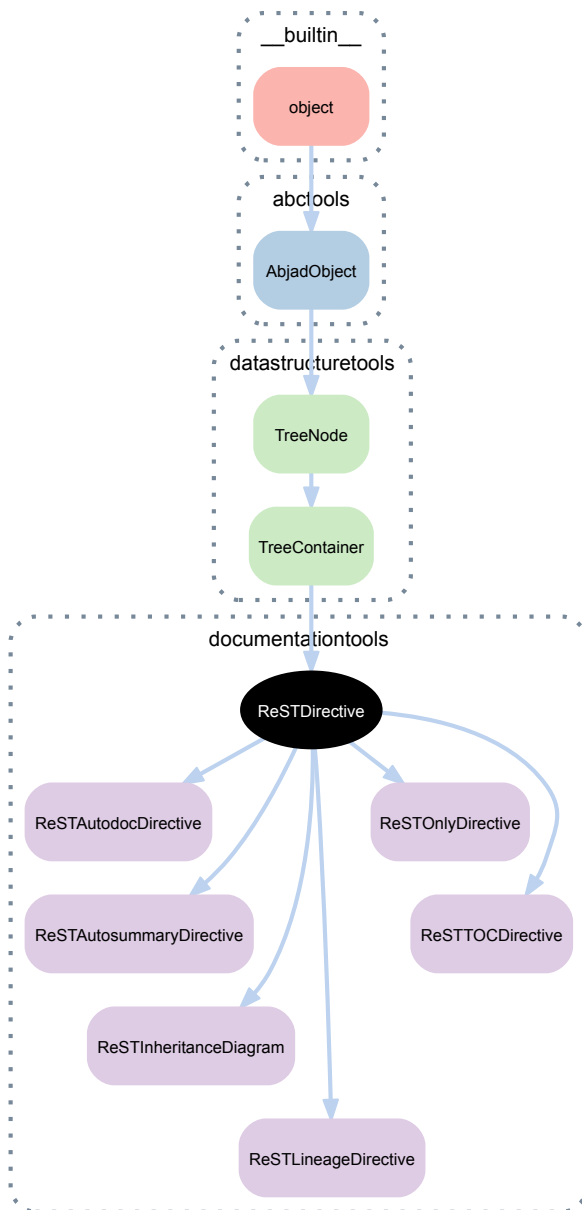
`GraphvizObject.attributes`

#### Special methods

- `(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.
- `(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.
- `(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.

Returns string.

### 54.1.3 documentationtools.ReSTDirective



**class** `documentationtools.ReSTDirective` (*argument=None, children=None, name=None, options=None*)

#### Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

### (TreeContainer).children

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

### (TreeNode).depth

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

### (TreeNode).depthwise\_inventory

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
```

```
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

`ReSTDirective.directive`

`(TreeNode).graph_order`

`(TreeNode).improper_parentage`

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

`(TreeContainer).leaves`

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

`ReSTDirective.node_class`

`(TreeContainer).nodes`

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```



```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

`ReSTDirective.options`

`(TreeNode).parent`

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`(TreeNode).proper_parentage`

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

`ReSTDirective.rest_format`

`(TreeNode).root`

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

`ReSTDirective.argument`

`(TreeNode).name`

## Methods

`(TreeContainer).append(node)`

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

`(TreeContainer).extend(expr)`

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**index** (*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) .**insert** (*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**pop** (*i*=-1)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns *node*.

(TreeContainer) .**remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

(TreeContainer) .**\_\_contains\_\_** (*expr*)

True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode) .**\_\_copy\_\_** (\*args)

(TreeNode) .**\_\_deepcopy\_\_** (\*args)

(TreeContainer) .**\_\_delitem\_\_** (i)

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(TreeContainer) .**\_\_eq\_\_** (expr)

True if type, duration and children are equivalent, otherwise False.

Returns boolean.

(TreeContainer) .**\_\_getitem\_\_** (i)

Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

```
(TreeNode).__getstate__()
```

```
(TreeContainer).__iter__()
```

```
(TreeContainer).__len__()
```

Returns nonnegative integer number of nodes in container.

```
(TreeNode).__ne__(expr)
```

```
(TreeNode).__repr__()
```

```
(TreeContainer).__setitem__(i, expr)
```

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

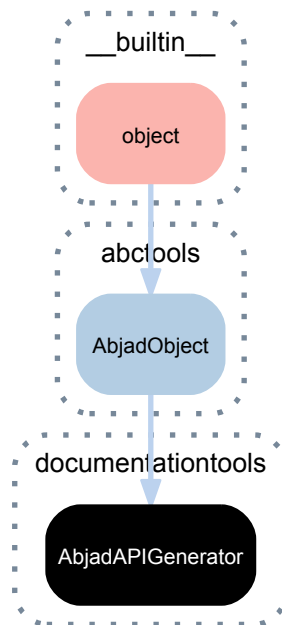
```
>>> a.children == (c,)
True
```

Return *None*.

```
(TreeNode).__setstate__(state)
```

## 54.2 Concrete classes

### 54.2.1 documentationtools.AbjadAPIGenerator



**class** `documentationtools.AbjadAPIGenerator`

Creates Abjad's API ReST:

- writes ReST pages for individual classes and functions
- writes the API index ReST
- handles sorting tools packages into composition, manual-loading and unstable
- handles ignoring private tools packages

Returns *AbjadAPIGenerator* instance.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`AbjadAPIGenerator.docs_api_index_path`  
Path to index.rst for Abjad API.

`AbjadAPIGenerator.package_prefix`

`AbjadAPIGenerator.path_definitions`  
Code path / docs path / package prefix triples.

`AbjadAPIGenerator.root_package`

`AbjadAPIGenerator.tools_package_path_index`

## Special methods

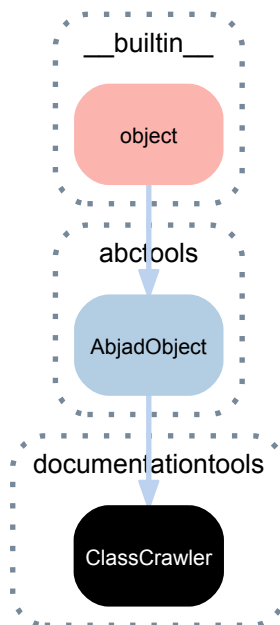
`AbjadAPIGenerator.__call__` (*verbose=False*)

`(AbjadObject).__eq__` (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__` (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__` ()  
Interpreter representation of Abjad object.  
Returns string.

## 54.2.2 documentationtools.ClassCrawler



```
class documentationtools.ClassCrawler (code_root,          include_private_objects=False,
                                       root_package_name=None)
```

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ClassCrawler.code_root`

`ClassCrawler.include_private_objects`

`ClassCrawler.module_crawler`

`ClassCrawler.root_package_name`



## Special methods

`ClassCrawler.__call__()`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

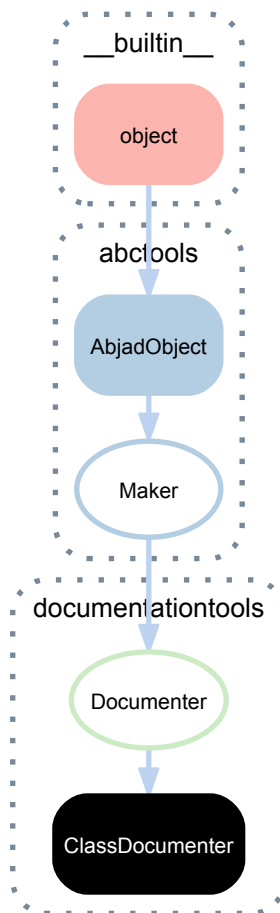
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 54.2.3 documentationtools.ClassDocumenter



**class** `documentationtools.ClassDocumenter` (*obj*, *prefix*=*'abjad.tools.'*)

Generates an ReST API entry for a given class:

```

>>> cls = documentationtools.ClassDocumenter
>>> documenter = documentationtools.ClassDocumenter(cls)
>>> restructured_text = documenter()
>>> print restructured_text
documentationtools.ClassDocumenter
=====

.. abjad-lineage:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter
  
```

```

.. autoclass:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter

.. only:: html

Attribute summary
-----

.. autosummary::

    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.class_methods
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.data
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.inherited_attributes
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.is_abstract
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.methods
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.module_name
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.new
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.object
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.prefix
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.readonly_properties
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.readwrite_properties
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.special_methods
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.static_methods
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.storage_format
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.write
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.__call__
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.__eq__
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.__ne__
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.__repr__

Bases
-----

- :py:class:`documentationtools.Documenter` <abjad.tools.documentationtools.Documenter.Documenter.Documenter>
- :py:class:`abctools.Maker` <abjad.tools.abctools.Maker.Maker.Maker>
- :py:class:`abctools.AbjadObject` <abjad.tools.abctools.AbjadObject.AbjadObject.AbjadObject>
- :py:class:`__builtin__.object` <object>

Read-only properties
-----

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.class_name
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.data
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.inherited_attributes
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.is_abstract
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.methods
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.module_name
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.object
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.prefix
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.readonly_properties
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.readwrite_properties
:noindex:

```

```

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.special
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.static
:noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.storage
:noindex:

Methods
-----

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.new
:noindex:

Static methods
-----

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.write
:noindex:

Special methods
-----

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.__call__
:noindex:

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.__eq__
:noindex:

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.__ne__
:noindex:

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.ClassDocumenter.__repr__
:noindex:

```

Returns `ClassDocumenter` instance.

## Bases

- `documentationtools.Documenter`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ClassDocumenter.class_methods`

`ClassDocumenter.data`

`ClassDocumenter.inherited_attributes`

`ClassDocumenter.is_abstract`

`ClassDocumenter.methods`

`(Documenter).module_name`

`(Documenter).object`

`(Documenter).prefix`

`ClassDocumenter.readonly_properties`

`ClassDocumenter.readwrite_properties`

`ClassDocumenter.special_methods`

`ClassDocumenter.static_methods`

`(Maker).storage_format`

Storage format of maker.

Returns string.

## Methods

`(Documenter).new(obj=None, prefix=None)`

## Static methods

`(Documenter).write(file_path, restructured_text)`

## Special methods

`ClassDocumenter.__call__()`

Generate documentation.

Returns string.

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

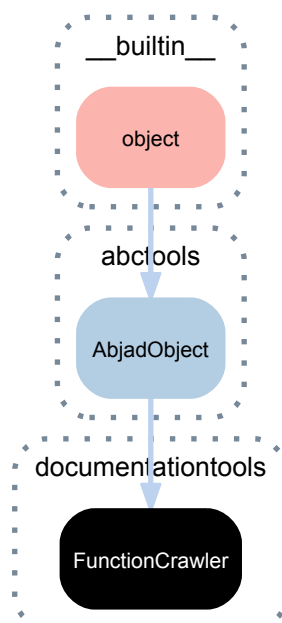
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 54.2.4 documentationtools.FunctionCrawler



```
class documentationtools.FunctionCrawler (code_root, include_private_objects=False,  
                                           root_package_name=None)
```

### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

`FunctionCrawler.code_root`

`FunctionCrawler.include_private_objects`

`FunctionCrawler.module_crawler`

`FunctionCrawler.root_package_name`

### Special methods

`FunctionCrawler.__call__()`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

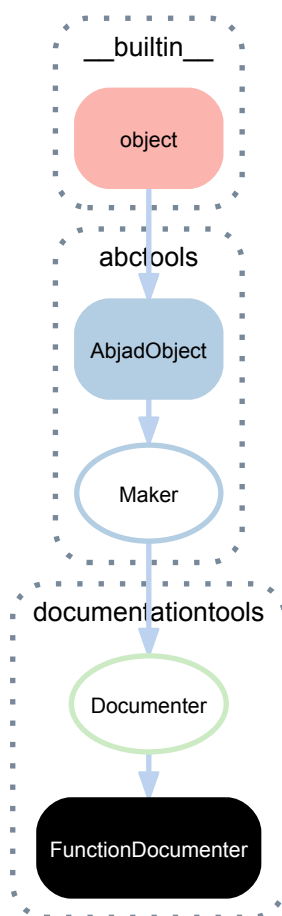
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

## 54.2.5 documentationtools.FunctionDocumenter



**class** `documentationtools.FunctionDocumenter` (*obj*, *prefix*='abjad.tools.')

FunctionDocumenter generates an ReST entry for a given function:

```
>>> documenter = documentationtools.FunctionDocumenter(notetools.make_notes)
>>> print documenter()
notetools.make_notes
=====

.. autofunction:: abjad.tools.notetools.make_notes.make_notes
```

Returns `FunctionDocumenter` `instance`.

## Bases

- `documentationtools.Documenter`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(Documenter).module_name`

`(Documenter).object`

`(Documenter).prefix`

(Maker) **.storage\_format**  
Storage format of maker.  
Returns string.

## Methods

(Documenter) **.new** (*obj=None, prefix=None*)

## Static methods

(Documenter) **.write** (*file\_path, restructured\_text*)

## Special methods

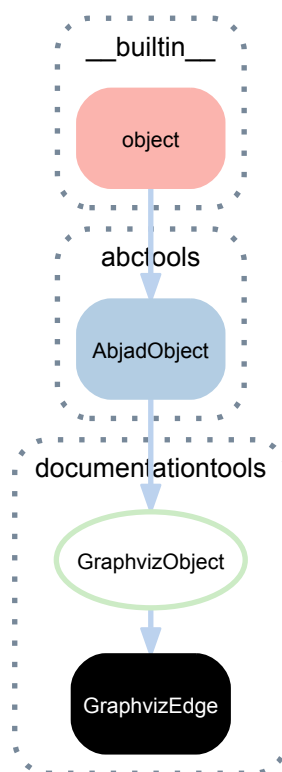
FunctionDocumenter **.\_\_call\_\_**()  
Generate documentation.  
Returns string.

(AbjadObject) **.\_\_eq\_\_** (*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(AbjadObject) **.\_\_repr\_\_**()  
Interpreter representation of Abjad object.  
Returns string.

### 54.2.6 documentationtools.GraphvizEdge



**class** `documentationtools.GraphvizEdge` (*attributes=None, is\_directed=True*)  
A Graphviz edge.

#### Bases

- `documentationtools.GraphvizObject`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(GraphvizObject).attributes`

`GraphvizEdge.head`

`GraphvizEdge.tail`

#### Read/write properties

`GraphvizEdge.is_directed`

#### Special methods

`GraphvizEdge.__call__(*args)`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

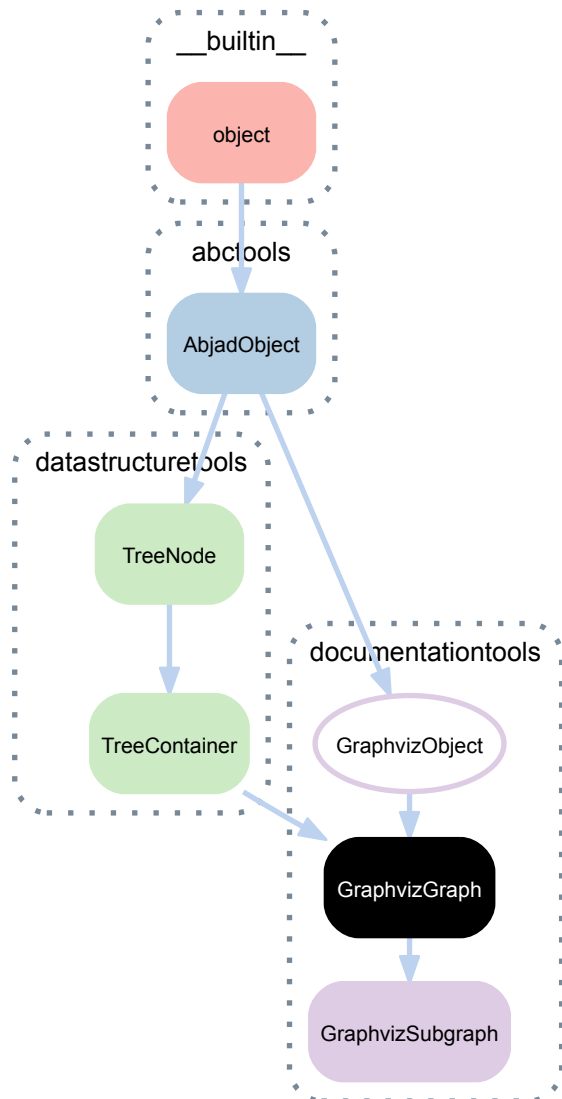
Returns boolean.



(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 54.2.7 documentationtools.GraphvizGraph



```
class documentationtools.GraphvizGraph (attributes=None, children=None,
                                         edge_attributes=None, is_digraph=True,
                                         name=None, node_attributes=None)
```

Abjad model of a Graphviz graph:

```
>>> graph = documentationtools.GraphvizGraph(name='G')
```

Create other graphviz objects to insert into the graph:

```
>>> cluster_0 = documentationtools.GraphvizSubgraph(name='0')
>>> cluster_1 = documentationtools.GraphvizSubgraph(name='1')
>>> a0 = documentationtools.GraphvizNode(name='a0')
>>> a1 = documentationtools.GraphvizNode(name='a1')
```

```
>>> a2 = documentationtools.GraphvizNode(name='a2')
>>> a3 = documentationtools.GraphvizNode(name='a3')
>>> b0 = documentationtools.GraphvizNode(name='b0')
>>> b1 = documentationtools.GraphvizNode(name='b1')
>>> b2 = documentationtools.GraphvizNode(name='b2')
>>> b3 = documentationtools.GraphvizNode(name='b3')
>>> start = documentationtools.GraphvizNode(name='start')
>>> end = documentationtools.GraphvizNode(name='end')
```

Group objects together into a tree:

```
>>> graph.extend([cluster_0, cluster_1, start, end])
>>> cluster_0.extend([a0, a1, a2, a3])
>>> cluster_1.extend([b0, b1, b2, b3])
```

Connect objects together with edges:

```
>>> edge = documentationtools.GraphvizEdge()(start, a0)
>>> edge = documentationtools.GraphvizEdge()(start, b0)
>>> edge = documentationtools.GraphvizEdge()(a0, a1)
>>> edge = documentationtools.GraphvizEdge()(a1, a2)
>>> edge = documentationtools.GraphvizEdge()(a1, b3)
>>> edge = documentationtools.GraphvizEdge()(a2, a3)
>>> edge = documentationtools.GraphvizEdge()(a3, a0)
>>> edge = documentationtools.GraphvizEdge()(a3, end)
>>> edge = documentationtools.GraphvizEdge()(b0, b1)
>>> edge = documentationtools.GraphvizEdge()(b1, b2)
>>> edge = documentationtools.GraphvizEdge()(b2, b3)
>>> edge = documentationtools.GraphvizEdge()(b2, a3)
>>> edge = documentationtools.GraphvizEdge()(b3, end)
```

Add attributes to style the objects:

```
>>> cluster_0.attributes['style'] = 'filled'
>>> cluster_0.attributes['color'] = 'lightgrey'
>>> cluster_0.attributes['label'] = 'process #1'
>>> cluster_0.node_attributes['style'] = 'filled'
>>> cluster_0.node_attributes['color'] = 'white'
>>> cluster_1.attributes['color'] = 'blue'
>>> cluster_1.attributes['label'] = 'process #2'
>>> cluster_1.node_attributes['style'] = ('filled', 'rounded')
>>> start.attributes['shape'] = 'Mdiamond'
>>> end.attributes['shape'] = 'Msquare'
```

Access the computed graphviz format of the graph:

```
>>> print graph.graphviz_format
digraph G {
    subgraph cluster_0 {
        graph [color=lightgrey,
            label="process #1",
            style=filled];
        node [color=white,
            style=filled];
        a0;
        a1;
        a2;
        a3;
        a0 -> a1;
        a1 -> a2;
        a2 -> a3;
        a3 -> a0;
    }
    subgraph cluster_1 {
        graph [color=blue,
            label="process #2"];
        node [style="filled, rounded"];
        b0;
        b1;
        b2;
        b3;
        b0 -> b1;
```

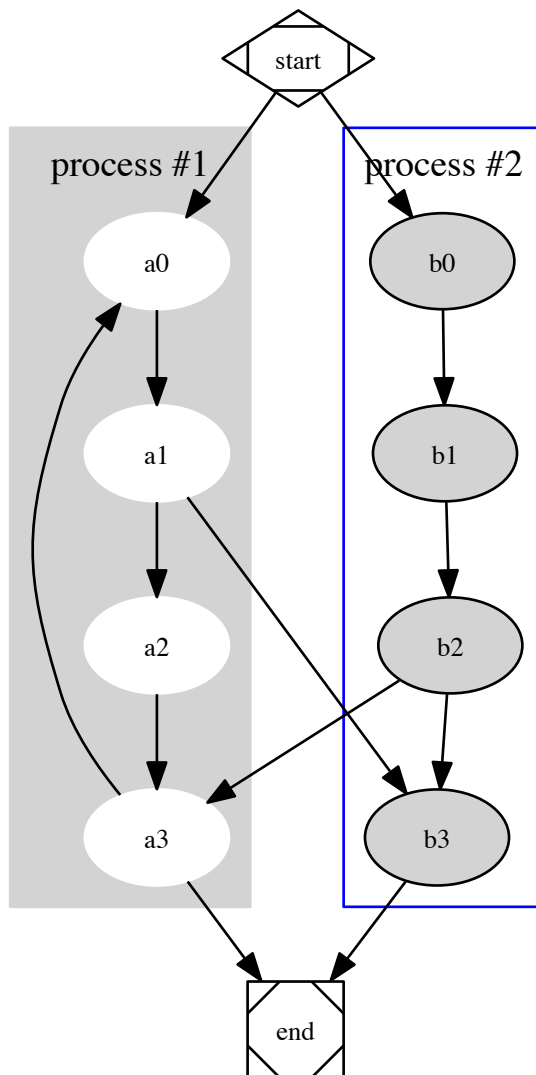
```

    b1 -> b2;
    b2 -> b3;
}
start [shape=Mdiamond];
end [shape=Msquare];
a1 -> b3;
a3 -> end;
b2 -> a3;
b3 -> end;
start -> a0;
start -> b0;
}

```

View the graph:

```
>>> iotools.graph(graph)
```



Graphs can also be created without defining names. Canonical names will be automatically determined for all members whose *name* is *None*:

```

>>> graph = documentationtools.GraphvizGraph()
>>> graph.append(documentationtools.GraphvizSubgraph())
>>> graph[0].append(documentationtools.GraphvizNode())
>>> graph[0].append(documentationtools.GraphvizNode())
>>> graph[0].append(documentationtools.GraphvizNode())
>>> graph[0].append(documentationtools.GraphvizSubgraph())
>>> graph[0][-1].append(documentationtools.GraphvizNode())
>>> graph.append(documentationtools.GraphvizNode())
>>> edge = documentationtools.GraphvizEdge()(graph[0][1], graph[1])

```

```
>>> edge = documentationtools.GraphvizEdge() (
...     graph[0][0], graph[0][-1][0])
```

```
>>> print graph.graphviz_format
digraph Graph {
  subgraph cluster_0 {
    node_0_0;
    node_0_1;
    node_0_2;
    subgraph cluster_0_3 {
      node_0_3_0;
    }
    node_0_0 -> node_0_3_0;
  }
  node_1;
  node_0_1 -> node_1;
}
```

Returns GraphvizGraph instance.

## Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `documentationtools.GraphvizObject`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(GraphvizObject).**attributes**

GraphvizGraph.**canonical\_name**

(TreeContainer).**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

(TreeNode).**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(TreeNode). **depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

GraphvizGraph. **edge\_attributes**

(TreeNode). **graph\_order**

GraphvizGraph. **graphviz\_format**

(TreeNode). **improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeContainer) **.leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

GraphvizGraph **.node\_attributes**

(TreeContainer) **.nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(TreeNode) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(*TreeNode*) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

GraphvizGraph.**unflattened\_graphviz\_format**

## Read/write properties

GraphvizGraph.**is\_digraph**

(*TreeNode*) **.name**

## Methods

(*TreeContainer*) **.append**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.extend** (*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.index** (*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) **.insert** (*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
```



```
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.pop** (*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) **.remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
        TreeNode()
    )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

(TreeContainer).**\_\_contains\_\_**(*expr*)  
True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode).**\_\_copy\_\_**(\*args)

(TreeNode).**\_\_deepcopy\_\_**(\*args)

(TreeContainer).**\_\_delitem\_\_**(*i*)  
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(TreeContainer).**\_\_eq\_\_**(*expr*)  
True if type, duration and children are equivalent, otherwise False.

Returns boolean.

(TreeContainer).**\_\_getitem\_\_**(*i*)  
Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(*TreeNode*).**\_\_getstate\_\_**()

(*TreeContainer*).**\_\_iter\_\_**()

(*TreeContainer*).**\_\_len\_\_**()

Returns nonnegative integer number of nodes in container.

(*TreeNode*).**\_\_ne\_\_**(*expr*)

(*TreeNode*).**\_\_repr\_\_**()

(*TreeContainer*).**\_\_setitem\_\_**(*i*, *expr*)

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

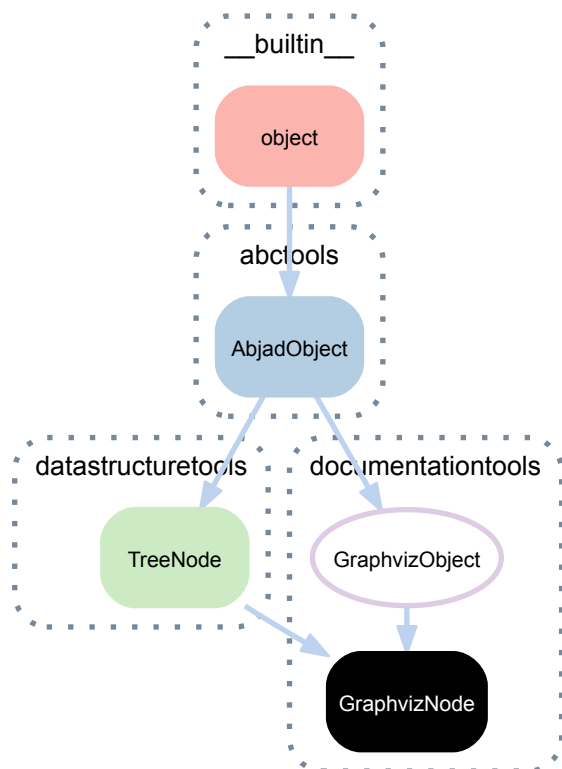
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

```
(TreeNode).__setstate__(state)
```

## 54.2.8 documentationtools.GraphvizNode



```
class documentationtools.GraphvizNode (attributes=None, name=None)
```

### Bases

- `datastructuretools.TreeNode`
- `documentationtools.GraphvizObject`
- `abctools.AbjadObject`
- `__builtin__.object`

### Read-only properties

```
(GraphvizObject).attributes
```

```
GraphvizNode.canonical_name
```

```
(TreeNode).depth
```

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

#### (TreeNode).**depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

#### GraphvizNode.**edges**

#### (TreeNode).**graph\_order**

#### (TreeNode).**improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(*TreeNode*) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

(TreeNode) .name

## Special methods

(TreeNode) .\_\_copy\_\_ (\*args)

(TreeNode) .\_\_deepcopy\_\_ (\*args)

(TreeNode) .\_\_eq\_\_ (expr)

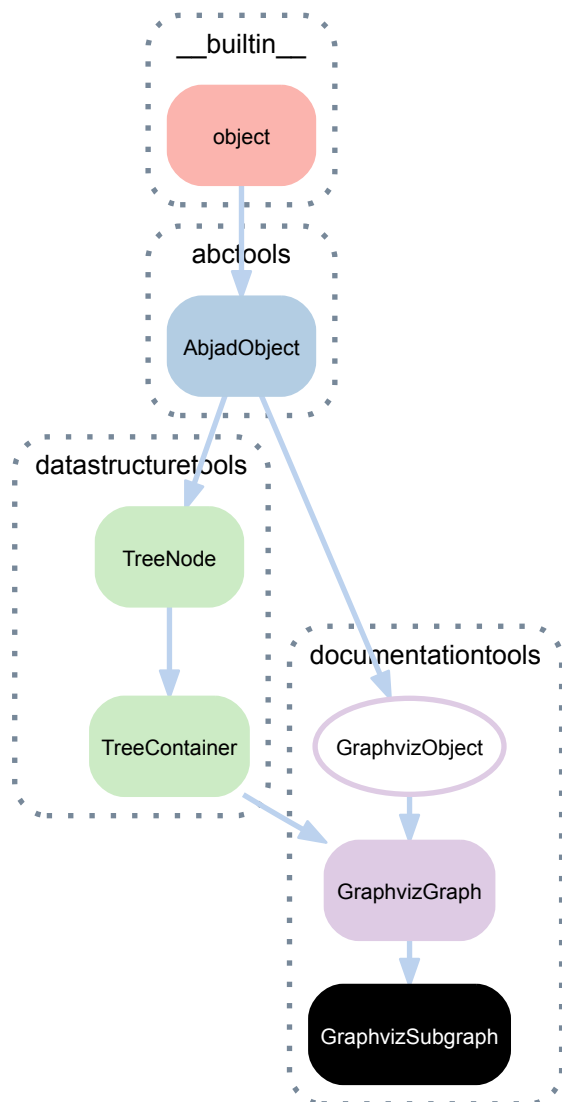
(TreeNode) .\_\_getstate\_\_ ()

(TreeNode) .\_\_ne\_\_ (expr)

(TreeNode) .\_\_repr\_\_ ()

(TreeNode) .\_\_setstate\_\_ (state)

## 54.2.9 documentationtools.GraphvizSubgraph



```
class documentationtools.GraphvizSubgraph (attributes=None,          children=None,
                                           edge_attributes=None,      is_cluster=True,
                                           name=None, node_attributes=None)
```

A Graphviz cluster subgraph.

## Bases

- `documentationtools.GraphvizGraph`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `documentationtools.GraphvizObject`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(`GraphvizObject`).**attributes**

`GraphvizSubgraph`.**canonical\_name**

(`TreeContainer`).**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

(`TreeNode`).**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.



**(TreeNode).depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

**(GraphvizGraph).edge\_attributes****GraphvizSubgraph.edges****(TreeNode).graph\_order****(GraphvizGraph).graphviz\_format****(TreeNode).improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

**(TreeContainer).leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
```

```
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

(GraphvizGraph) **.node\_attributes**

(TreeContainer) **.nodes**

The collection of *TreeNode*s produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(TreeNode) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(*TreeNode*) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

(*GraphvizGraph*) **.unflattened\_graphviz\_format**

## Read/write properties

*GraphvizSubgraph*.**is\_cluster**

(*GraphvizGraph*) **.is\_digraph**

(*TreeNode*) **.name**

## Methods

(*TreeContainer*) **.append**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer (
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer (
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**extend** (*expr*)  
 Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer (
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**index** (*node*)  
 Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) .**insert** (*i*, *node*)  
 Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer (
  children=(
```

```

        TreeNode(),
        TreeNode()
    )
)

```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)

```

Return *None*.

(TreeContainer) **.pop** (*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()

```

```
>>> a.extend([b, c])

```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)

```

```
>>> node = a.pop()

```

```
>>> node == c
True

```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)

```

Returns node.

(TreeContainer) **.remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()

```

```
>>> a.extend([b, c])

```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)

```

```
>>> a.remove(b)

```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

`(TreeContainer).__contains__(expr)`  
 True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

`(TreeNode).__copy__(*args)`

`(TreeNode).__deepcopy__(*args)`

`(TreeContainer).__delitem__(i)`  
 Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

`(TreeContainer).__eq__(expr)`  
 True if type, duration and children are equivalent, otherwise False.

Returns boolean.

`(TreeContainer).__getitem__(i)`  
 Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(*TreeNode*) **.\_\_getstate\_\_**()

(*TreeContainer*) **.\_\_iter\_\_**()

(*TreeContainer*) **.\_\_len\_\_**()

Returns nonnegative integer number of nodes in container.

(*TreeNode*) **.\_\_ne\_\_**(*expr*)

(*TreeNode*) **.\_\_repr\_\_**()

(*TreeContainer*) **.\_\_setitem\_\_**(*i*, *expr*)

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

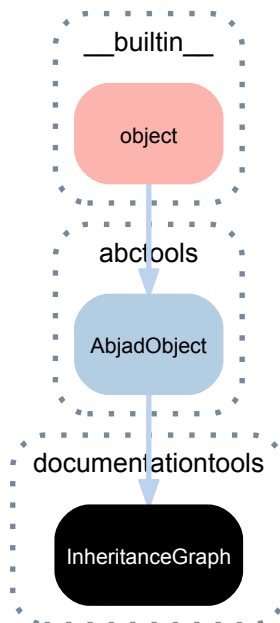
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

```
(TreeNode).__setstate__(state)
```

## 54.2.10 documentationtools.InheritanceGraph



```
class documentationtools.InheritanceGraph(addresses=('abjad', ), lineage_addresses=None, lineage_prune_distance=None, recurse_into_submodules=True, root_addresses=None, use_clusters=True, use_groups=True)
```

Generates a graph of a class or collection of classes as a dictionary of parent-children relationships:

```
>>> class A(object): pass
...
>>> class B(A): pass
...
>>> class C(B): pass
...
>>> class D(B): pass
...
>>> class E(C, D): pass
...
>>> class F(A): pass
...
```

```
>>> graph = documentationtools.InheritanceGraph(addresses=(F, E))
```

`InheritanceGraph` may be instantiated from one or more instances, classes or modules. If instantiated from a module, all public classes in that module will be taken into the graph.

A `root_class` keyword may be defined at instantiation, which filters out all classes from the graph which do not inherit from that `root_class` (or are not already the `root_class`):

```
>>> graph = documentationtools.InheritanceGraph(
...     (A, B, C, D, E, F), root_addresses=(B,))
```

The class is intended for use in documenting packages.



To document all of Abjad, use this formulation:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',),)
```

To document only those classes descending from Container, use this formulation:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',),
...     root_addresses=(Container,),
...     )
```

To document only those classes whose lineage pass through componenttools, use this formulation:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',),
...     lineage_addresses=(componenttools,),
...     )
```

When creating the Graphviz representation, classes in the inheritance graph may be hidden, based on their distance from any defined lineage class:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',),
...     lineage_addresses=(marktools.Mark,),
...     lineage_prune_distance=1,
...     )
```

Returns InheritanceGraph instance.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`InheritanceGraph.addresses`

`InheritanceGraph.child_parents_mapping`

`InheritanceGraph.graphviz_format`

`InheritanceGraph.graphviz_graph`

`InheritanceGraph.immediate_classes`

`InheritanceGraph.lineage_addresses`

`InheritanceGraph.lineage_classes`

`InheritanceGraph.lineage_distance_mapping`

`InheritanceGraph.lineage_prune_distance`

`InheritanceGraph.parent_children_mapping`

`InheritanceGraph.recurse_into_submodules`

`InheritanceGraph.root_addresses`

`InheritanceGraph.root_classes`

`InheritanceGraph.use_clusters`

`InheritanceGraph.use_groups`

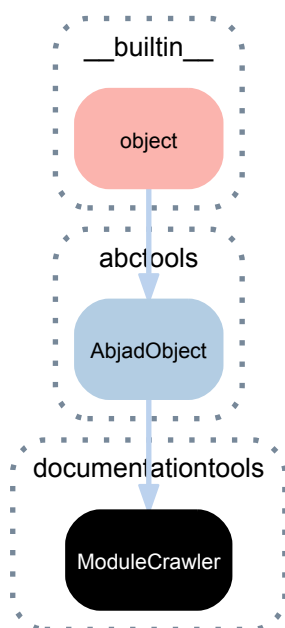
## Special methods

(AbjadObject) .\_\_eq\_\_(*expr*)  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

(AbjadObject) . **\_\_ne\_\_** (*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

### 54.2.11 documentationtools.ModuleCrawler



```
class documentationtools.ModuleCrawler (code_root='', ignored_directory_names=('test',
                                                                              'svn', '__pycache__'), root_package_name=None,
                                          visit_private_modules=False)
    Crawls code_root, yielding all module objects whose name begins with root_package_name.
    Return ModuleCrawler instance.
```

## Bases

- `abctools.AbjadObject`
- `builtin .object`

## Read-only properties

ModuleCrawler.**code\_root**

ModuleCrawler.ignored\_directory\_names

ModuleCrawler.root\_package\_name

ModuleCrawler.visit\_private\_modules

## Special methods

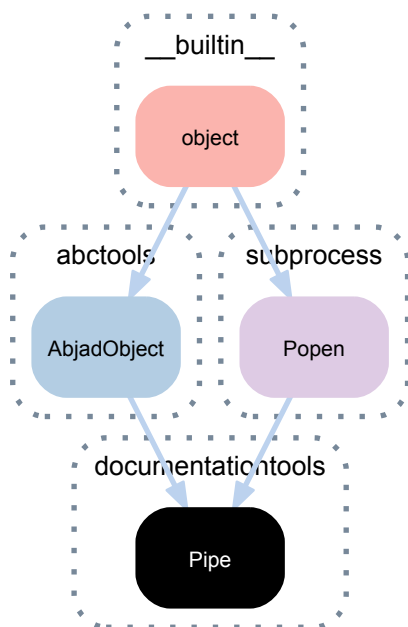
(AbjadObject).**\_\_eq\_\_**(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

ModuleCrawler.**\_\_iter\_\_**()

(AbjadObject).**\_\_ne\_\_**(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

### 54.2.12 documentationtools.Pipe



**class** documentationtools.**Pipe** (*executable='python', arguments=['-i'], timeout=0*)  
 A two-way, non-blocking pipe for interprocess communication:

```
>>> pipe = documentationtools.Pipe('python', ['-i'])
>>> pipe.writeline('my_list = [1, 2, 3]')
>>> pipe.writeline('print my_list')
```

Return *Pipe* instance.

## Bases

- abctools.AbjadObject
- subprocess.Popen
- \_\_builtin\_\_.object

## Read-only properties

`Pipe.arguments`

`Pipe.executable`

`Pipe.timeout`

## Methods

`Pipe.close()`

Close the pipe.

`(Popen).communicate(input=None)`

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional input argument should be a string to be sent to the child process, or None, if no data should be sent to the child.

`communicate()` returns a tuple (stdout, stderr).

`(Popen).kill()`

Kill the process with SIGKILL

`(Popen).poll()`

`Pipe.read()`

Read from the pipe.

`Pipe.read_wait(seconds=0.01)`

Try to read from the pipe. Wait *seconds* if nothing comes out, and repeat.

Should be used with caution, as this may loop forever.

`(Popen).send_signal(sig)`

Send a signal to the process

`(Popen).terminate()`

Terminate the process with SIGTERM

`(Popen).wait()`

Wait for child process to terminate. Returns returncode attribute.

`Pipe.write(data)`

Write *data* into the pipe.

`Pipe.write_line(data)`

Write *data* into the pipe, terminated by a newline.

## Special methods

`(Popen).__del__(_maxint=9223372036854775807, _active=[])`

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

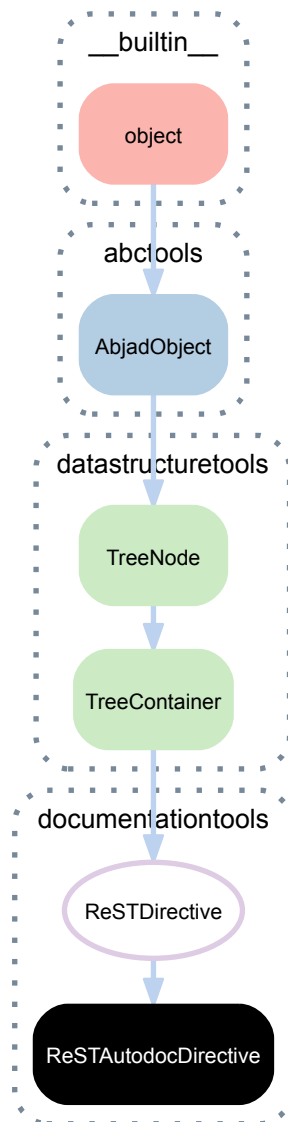
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 54.2.13 documentationtools.ReSTAutodocDirective



**class** `documentationtools.ReSTAutodocDirective` (*argument=None, children=None, directive=None, name=None, options=None*)

An ReST autodoc directive:

```

>>> autodoc = documentationtools.ReSTAutodocDirective(
...     argument='abjad.tools.spannertools.BeamSpanner.BeamSpanner.BeamSpanner',
...     directive='autoclass',
... )
>>> autodoc.options['noindex'] = True
>>> autodoc
ReSTAutodocDirective(
  argument='abjad.tools.spannertools.BeamSpanner.BeamSpanner.BeamSpanner',
  directive='autoclass',
  options={
    'noindex': True
  }
)
  
```

```

>>> print autodoc.rest_format
.. autoclass:: abjad.tools.spannertools.BeamSpanner.BeamSpanner.BeamSpanner
   :noindex:
  
```

Return *ReSTAutodocDirective* instance.

## Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(`TreeContainer`) **.children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

(`TreeNode`) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(`TreeNode`) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

(TreeNode) **.graph\_order**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeContainer) **.leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

(ReSTDirective) **.node\_class**

**(TreeContainer).nodes**

The collection of *TreeNode*s produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

**(ReSTDirective).options****(TreeNode).parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

**(TreeNode).proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```



```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(ReSTDirective) **.rest\_format**

(TreeNode) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

(ReSTDirective) **.argument**

ReSTAutodocDirective **.directive**

(TreeNode) **.name**

## Methods

(TreeContainer) **.append** (*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
)
)
```

Return *None*.

(TreeContainer) **.extend** (*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.index** (*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) **.insert** (*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

```
)
)
```

Return *None*.

(TreeContainer) .**pop** (*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) .**remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

(TreeContainer) .**\_\_contains\_\_** (*expr*)

True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode) .**\_\_copy\_\_** (\*args)

(TreeNode) .**\_\_deepcopy\_\_** (\*args)

(TreeContainer) .**\_\_delitem\_\_** (i)

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(TreeContainer) .**\_\_eq\_\_** (expr)

True if type, duration and children are equivalent, otherwise False.

Returns boolean.

(TreeContainer) .**\_\_getitem\_\_** (i)

Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(TreeNode).**\_\_getstate\_\_**()

(TreeContainer).**\_\_iter\_\_**()

(TreeContainer).**\_\_len\_\_**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**\_\_ne\_\_**(*expr*)

(TreeNode).**\_\_repr\_\_**()

(TreeContainer).**\_\_setitem\_\_**(*i*, *expr*)

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

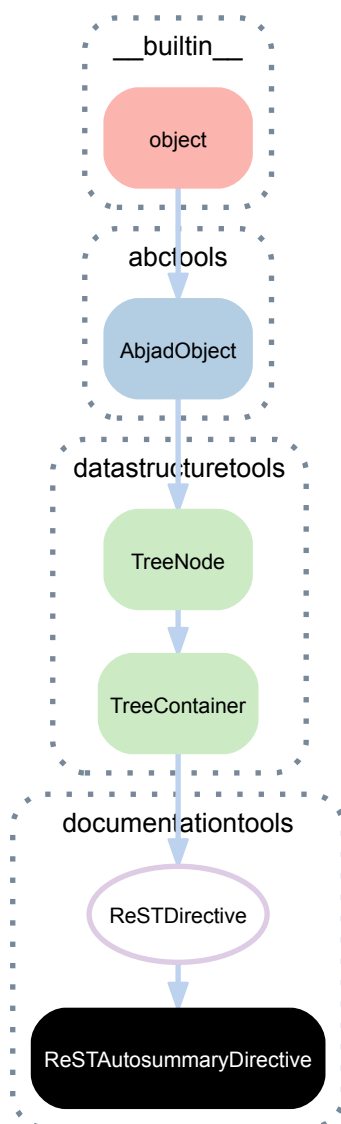
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

(TreeNode).**\_\_setstate\_\_**(*state*)

## 54.2.14 documentationtools.ReSTAutosummaryDirective



**class** `documentationtools.ReSTAutosummaryDirective` (*argument=None, children=None, name=None, options=None*)

An ReST AutosummaryTree directive:

```

>>> toc = documentationtools.ReSTAutosummaryDirective()
>>> for item in ['foo.Foo', 'bar.Bar', 'baz.Baz']:
...     toc.append(documentationtools.ReSTAutosummaryItem(text=item))
...
>>> toc
ReSTAutosummaryDirective(
  children=(
    ReSTAutosummaryItem(
      text='foo.Foo'
    ),
    ReSTAutosummaryItem(
      text='bar.Bar'
    ),
    ReSTAutosummaryItem(
      text='baz.Baz'
    )
  )
)
  
```

```
>>> print toc.rest_format
.. autosummary::

    foo.Foo
    bar.Bar
    baz.Baz
```

Return *ReSTAutosummaryDirective* instance.

## Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(*TreeContainer*) **.children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

**(TreeNode).depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

**ReSTAutosummaryDirective.directive****(TreeNode).graph\_order****(TreeNode).improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

**(TreeContainer).leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```



```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

ReSTAutosummaryDirective.**node\_class**

(TreeContainer).**nodes**

The collection of *TreeNode*s produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(ReSTDirective).**options**

(TreeNode).**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

**(TreeNode) .proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

**(ReSTDirective) .rest\_format****(TreeNode) .root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

**Read/write properties****(ReSTDirective) .argument****(TreeNode) .name****Methods****(TreeContainer) .append (node)**

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
```

```
)
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**extend** (*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**index** (*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) .**insert** (*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**pop** (*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) .**remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

(TreeContainer).**\_\_contains\_\_**(*expr*)  
True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode).**\_\_copy\_\_**(\*args)

(TreeNode).**\_\_deepcopy\_\_**(\*args)

(TreeContainer).**\_\_delitem\_\_**(*i*)  
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(TreeContainer).**\_\_eq\_\_**(*expr*)  
True if type, duration and children are equivalent, otherwise False.

Returns boolean.

(TreeContainer).**\_\_getitem\_\_**(*i*)  
Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(TreeNode) .\_\_getstate\_\_()

(TreeContainer) .\_\_iter\_\_()

(TreeContainer) .\_\_len\_\_()

Returns nonnegative integer number of nodes in container.

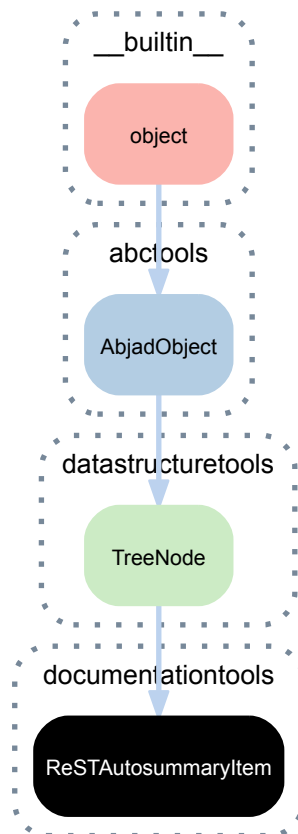
(TreeNode) .\_\_ne\_\_(*expr*)

(TreeNode) .\_\_repr\_\_()

ReSTAutosummaryDirective .\_\_setitem\_\_(*i*, *expr*)

(TreeNode) .\_\_setstate\_\_(*state*)

### 54.2.15 documentationtools.ReSTAutosummaryItem



**class** `documentationtools.ReSTAutosummaryItem` (*name=None, text=None*)

An ReST Autosummary item:

```
>>> item = documentationtools.ReSTAutosummaryItem(
...     text='abjad.tools.notetools.Note')
>>> item
ReSTAutosummaryItem(
    text='abjad.tools.notetools.Note'
)
```

```
>>> print item.rest_format
abjad.tools.notetools.Note
```

Return *ReSTAutosummaryItem* instance.

#### Bases

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

(*TreeNode*) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(TreeNode) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

(TreeNode) **.graph\_order**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```



Returns tuple of *TreeNode* instances.

(*TreeNode*) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(*TreeNode*) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

ReSTAutosummaryItem **.rest\_format**

(*TreeNode*) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

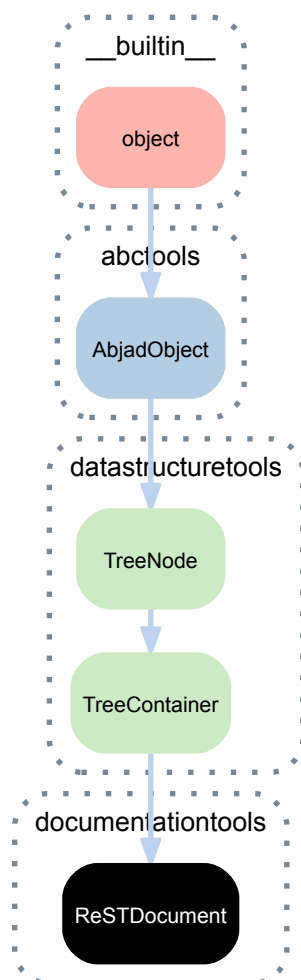
(*TreeNode*) **.name**

ReSTAutosummaryItem.**text**

## Special methods

```
(TreeNode).__copy__(*args)
(TreeNode).__deepcopy__(*args)
(TreeNode).__eq__(expr)
(TreeNode).__getstate__()
(TreeNode).__ne__(expr)
(TreeNode).__repr__()
(TreeNode).__setstate__(state)
```

## 54.2.16 documentationtools.ReSTDocument



**class** `documentationtools.ReSTDocument` (*children=None, name=None*)  
 An ReST document tree:

```
>>> document = documentationtools.ReSTDocument()
>>> document
ReSTDocument()
```

```

>>> document.append(documentationtools.ReSTHeading(
...     level=0, text='Hello World!'))
>>> document.append(documentationtools.ReSTParagraph(
...     text='blah blah blah'))
>>> toc = documentationtools.ReSTTOCDirective()
>>> toc.append('foo/bar')
>>> toc.append('bar/baz')
>>> toc.append('quux')
>>> document.append(toc)

```

```

>>> document
ReSTDocument(
  children=(
    ReSTHeading(
      level=0,
      text='Hello World!'
    ),
    ReSTParagraph(
      text='blah blah blah',
      wrap=True
    ),
    ReSTTOCDirective(
      children=(
        ReSTTOCItem(
          text='foo/bar'
        ),
        ReSTTOCItem(
          text='bar/baz'
        ),
        ReSTTOCItem(
          text='quux'
        )
      )
    )
  )
)

```

```

>>> print document.rest_format
#####
Hello World!
#####

blah blah blah

.. toctree::

    foo/bar
    bar/baz
    quux

```

Return *ReSTDocument* instance.

## Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(*TreeContainer*).**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(*TreeNode*) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
```

```
f
g
```

Returns dictionary.

(TreeNode) **.graph\_order**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeContainer) **.leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

ReSTDDocument **.node\_class**

(TreeContainer) **.nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(TreeNode) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(TreeNode) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

ReSTDDocument **.rest\_format**

(TreeNode) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

(TreeNode) **.name**

## Methods

(TreeContainer) **.append** (*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.extend** (*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**index** (*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) .**insert** (*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**pop** (*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```



```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) . **remove** (node)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

(TreeContainer) . **\_\_contains\_\_** (expr)

True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode) . **\_\_copy\_\_** (\*args)

(TreeNode) . **\_\_deepcopy\_\_** (\*args)

(TreeContainer) . **\_\_delitem\_\_** (i)

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(TreeContainer).**\_\_eq\_\_**(*expr*)

True if type, duration and children are equivalent, otherwise False.

Returns boolean.

(TreeContainer).**\_\_getitem\_\_**(*i*)

Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(TreeNode).**\_\_getstate\_\_**()

(TreeContainer).**\_\_iter\_\_**()

(TreeContainer).**\_\_len\_\_**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**\_\_ne\_\_**(*expr*)

```
(TreeNode).__repr__()
```

```
(TreeContainer).__setitem__(i, expr)
```

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

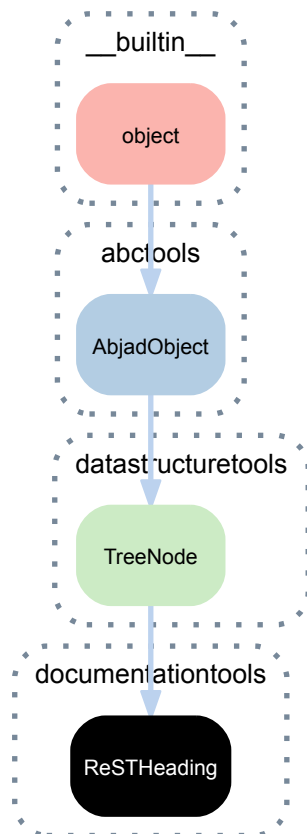
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

```
(TreeNode).__setstate__(state)
```

### 54.2.17 documentationtools.ReSTHeading



```
class documentationtools.ReSTHeading (level=0, name=None, text='')
    An ReST Heading:
```

```
>>> heading = documentationtools.ReSTHeading(
...     level=2, text='Section A')
>>> heading
ReSTHeading(
  level=2,
  text='Section A'
)
```

```
>>> print heading.rest_format
Section A
=====
```

Return *ReSTHeading* instance.

## Bases

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(*TreeNode*) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(*TreeNode*) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
```

```

DEPTH: 1
b
c
DEPTH: 2
d
e
f
g

```

Returns dictionary.

(TreeNode) **.graph\_order**

ReSTHeading **.heading\_characters**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()

```

```

>>> a.append(b)
>>> b.append(c)

```

```

>>> a.improper_parentage == (a,)
True

```

```

>>> b.improper_parentage == (b, a)
True

```

```

>>> c.improper_parentage == (c, b, a)
True

```

Returns tuple of *TreeNode* instances.

(TreeNode) **.parent**

The node's parent node:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()

```

```

>>> a.append(b)
>>> b.append(c)

```

```

>>> a.parent is None
True

```

```

>>> b.parent is a
True

```

```

>>> c.parent is b
True

```

Return *TreeNode* instance.

(TreeNode) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()

```

```

>>> a.append(b)
>>> b.append(c)

```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

`ReSTHeading.rest_format`

`(TreeNode).root`

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

`ReSTHeading.level`

`(TreeNode).name`

`ReSTHeading.text`

## Special methods

`(TreeNode).__copy__(*args)`

`(TreeNode).__deepcopy__(*args)`

`(TreeNode).__eq__(expr)`

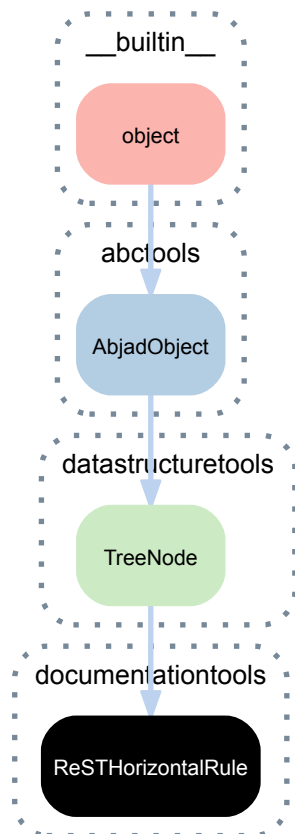
`(TreeNode).__getstate__()`

`(TreeNode).__ne__(expr)`

`(TreeNode).__repr__()`

`(TreeNode).__setstate__(state)`

### 54.2.18 documentationtools.ReSTHorizontalRule



**class** `documentationtools.ReSTHorizontalRule` (*name=None*)  
 An ReST horizontal rule:

```
>>> rule = documentationtools.ReSTHorizontalRule()
>>> rule
ReSTHorizontalRule()
```

```
>>> print rule.rest_format
-----
```

Return *ReSTHorizontalRule* instance.

#### Bases

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

(*TreeNode*) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(TreeNode) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

(TreeNode) **.graph\_order**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.



**(TreeNode) .parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

**(TreeNode) .proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

**ReSTHorizontalRule .rest\_format****(TreeNode) .root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

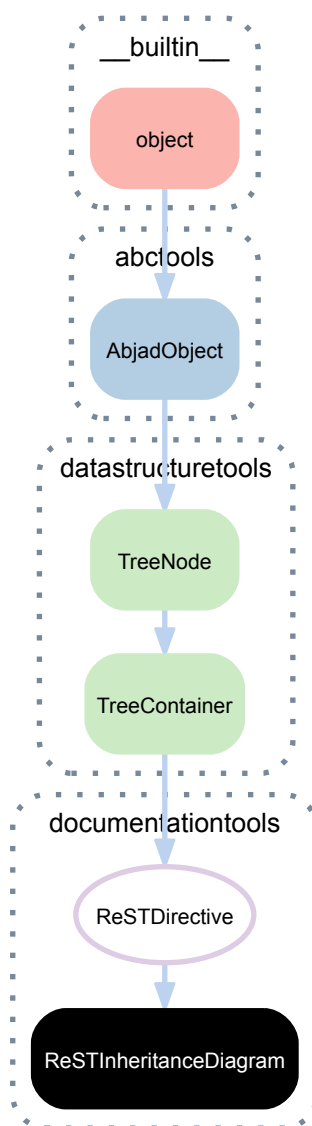
Return *TreeNode* instance.

**Read/write properties****(TreeNode) .name**

## Special methods

```
(TreeNode) .__copy__ (*args)
(TreeNode) .__deepcopy__ (*args)
(TreeNode) .__eq__ (expr)
(TreeNode) .__getstate__ ()
(TreeNode) .__ne__ (expr)
(TreeNode) .__repr__ ()
(TreeNode) .__setstate__ (state)
```

### 54.2.19 documentationtools.ReSTInheritanceDiagram



```
class documentationtools.ReSTInheritanceDiagram (argument=None,      children=None,
                                                name=None, options=None)
```

An ReST inheritance diagram directive:

```
>>> documentationtools.ReSTInheritanceDiagram(
...     argument=spannertools.BeamSpanner)
ReSTInheritanceDiagram(
```

```

argument='abjad.tools.spannertools.BeamSpanner.BeamSpanner.BeamSpanner',
options={
    'private-bases': True
}
)

```

```

>>> print _rest_format
.. inheritance-diagram:: abjad.tools.spannertools.BeamSpanner.BeamSpanner.BeamSpanner
:private-bases:

```

Return *ReSTInheritanceDiagram* instance.

## Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(*TreeContainer*) **.children**

The children of a *TreeContainer* instance:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()

```

```

>>> a.extend([b, c])
>>> b.extend([d, e])

```

```

>>> a.children == (b, c)
True

```

```

>>> b.children == (d, e)
True

```

```

>>> e.children == ()
True

```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.depth**

The depth of a node in a rhythm-tree structure:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)

```

```

>>> a.depth
0

```

```

>>> a[0].depth
1

```

```

>>> a[0][0].depth
2

```

Returns int.

(TreeNode) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

ReSTInheritanceDiagram **.directive**

(TreeNode) **.graph\_order**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeContainer) **.leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

(ReSTDirective) **.node\_class**

(TreeContainer) **.nodes**

The collection of *TreeNode*s produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(ReSTDirective) **.options**

(TreeNode) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

#### (TreeNode) .proper\_parentage

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

#### (ReSTDirective) .rest\_format

#### (TreeNode) .root

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

#### (ReSTDirective) .argument

#### (TreeNode) .name

## Methods

#### (TreeContainer) .append (node)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
```

```
)
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**extend** (*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**index** (*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) .**insert** (*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**pop** (*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) .**remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.



## Special methods

(TreeContainer).**\_\_contains\_\_**(*expr*)  
True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode).**\_\_copy\_\_**(\*args)

(TreeNode).**\_\_deepcopy\_\_**(\*args)

(TreeContainer).**\_\_delitem\_\_**(*i*)  
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(TreeContainer).**\_\_eq\_\_**(*expr*)  
True if type, duration and children are equivalent, otherwise False.

Returns boolean.

(TreeContainer).**\_\_getitem\_\_**(*i*)  
Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(*TreeNode*) .\_\_getstate\_\_()

(*TreeContainer*) .\_\_iter\_\_()

(*TreeContainer*) .\_\_len\_\_()

Returns nonnegative integer number of nodes in container.

(*TreeNode*) .\_\_ne\_\_(*expr*)

(*TreeNode*) .\_\_repr\_\_()

(*TreeContainer*) .\_\_setitem\_\_(*i, expr*)

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

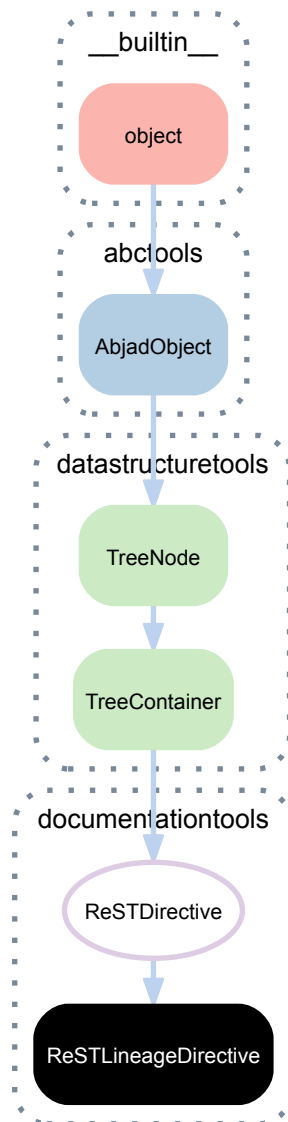
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

(*TreeNode*) .\_\_setstate\_\_(*state*)

### 54.2.20 documentationtools.ReSTLineageDirective



**class** `documentationtools.ReSTLineageDirective` (*argument=None, children=None, name=None, options=None*)

An ReST Abjad lineage diagram directive:

```
>>> documentationtools.ReSTLineageDirective(argument=spannertools.BeamSpanner)
ReSTLineageDirective(
    argument='abjad.tools.spannertools.BeamSpanner.BeamSpanner.BeamSpanner'
)
```

```
>>> print _.rest_format
.. abjad-lineage:: abjad.tools.spannertools.BeamSpanner.BeamSpanner.BeamSpanner
```

Return *ReSTLineageDirective* instance.

#### Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`

- `__builtin__.object`

## Read-only properties

(TreeContainer).**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

(TreeNode).**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(TreeNode).**depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
```

```

DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g

```

Returns dictionary.

`ReSTLineageDirective.directive`

`(TreeNode).graph_order`

`(TreeNode).improper_parentage`

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()

```

```

>>> a.append(b)
>>> b.append(c)

```

```

>>> a.improper_parentage == (a,)
True

```

```

>>> b.improper_parentage == (b, a)
True

```

```

>>> c.improper_parentage == (c, b, a)
True

```

Returns tuple of *TreeNode* instances.

`(TreeContainer).leaves`

The leaves of a *TreeContainer* instance:

```

>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')

```

```

>>> a.extend([b, c])
>>> b.extend([d, e])

```

```

>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c

```

Returns tuple.

`(ReSTDirective).node_class`

`(TreeContainer).nodes`

The collection of *TreeNodes* produced by iterating a node depth-first:

```

>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()

```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(ReSTDirective).**options**

(TreeNode).**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(TreeNode).**proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(ReSTDirective) .**rest\_format**

(TreeNode) .**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

(ReSTDirective) .**argument**

(TreeNode) .**name**

## Methods

(TreeContainer) .**append** (*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**extend** (*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**index** (node)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) .**insert** (i, node)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) .**pop** (i=-1)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```



```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) .**remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

(TreeContainer) .**\_\_contains\_\_** (*expr*)

True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode) .**\_\_copy\_\_** (\*args)

(TreeNode) .**\_\_deepcopy\_\_** (\*args)

(TreeContainer) .**\_\_delitem\_\_** (i)

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(TreeContainer) .**\_\_eq\_\_** (expr)

True if type, duration and children are equivalent, otherwise False.

Returns boolean.

(TreeContainer) .**\_\_getitem\_\_** (i)

Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

```
(TreeNode).__getstate__()
```

```
(TreeContainer).__iter__()
```

```
(TreeContainer).__len__()
```

Returns nonnegative integer number of nodes in container.

```
(TreeNode).__ne__(expr)
```

```
(TreeNode).__repr__()
```

```
(TreeContainer).__setitem__(i, expr)
```

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

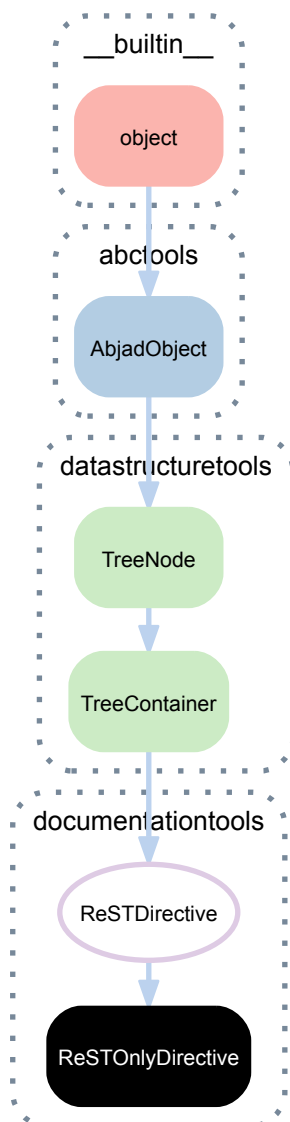
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

```
(TreeNode).__setstate__(state)
```

## 54.2.21 documentationtools.ReSTOnlyDirective



**class** `documentationtools.ReSTOnlyDirective` (*argument=None*, *children=None*,  
*name=None*)

An ReST *only* directive:

```
>>> only = documentationtools.ReSTOnlyDirective(argument='latex')
```

```
>>> heading = documentationtools.ReSTHeading(
...     level=3, text='A LaTeX-Only Heading')
>>> only.append(heading)
>>> only
ReSTOnlyDirective(
  argument='latex',
  children=(
    ReSTHeading(
      level=3,
      text='A LaTeX-Only Heading'
    ),
  )
)
```

```
>>> print only.rest_format
.. only:: latex

A LaTeX-Only Heading
```

Return *ReSTOnlyDirective* instance.

## Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`(TreeContainer).children`

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

`(TreeNode).depth`

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

`(TreeNode).depthwise_inventory`

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

`ReSTOnlyDirective.directive`

`(TreeNode).graph_order`

`(TreeNode).improper_parentage`

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

`(TreeContainer).leaves`

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

(ReSTDirective) **.node\_class**

(TreeContainer) **.nodes**

The collection of *TreeNode*s produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

(ReSTDirective) **.options**

(TreeNode) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(TreeNode) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(ReSTDirective) **.rest\_format**

(TreeNode) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

(ReSTDirective) **.argument**

(TreeNode) **.name**

## Methods

(TreeContainer) **.append** (*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```



```
>>> a.append(c)
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.extend** (*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.index** (*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) **.insert** (*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.pop** (*i*=-1)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Returns node.

(TreeContainer) **.remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

`(TreeContainer).__contains__(expr)`  
 True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

`(TreeNode).__copy__(*args)`

`(TreeNode).__deepcopy__(*args)`

`(TreeContainer).__delitem__(i)`  
 Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

`(TreeContainer).__eq__(expr)`  
 True if type, duration and children are equivalent, otherwise False.

Returns boolean.

`(TreeContainer).__getitem__(i)`  
 Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

(*TreeNode*) .\_\_getstate\_\_()

(*TreeContainer*) .\_\_iter\_\_()

(*TreeContainer*) .\_\_len\_\_()

Returns nonnegative integer number of nodes in container.

(*TreeNode*) .\_\_ne\_\_(*expr*)

(*TreeNode*) .\_\_repr\_\_()

(*TreeContainer*) .\_\_setitem\_\_(*i, expr*)

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice *i*. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

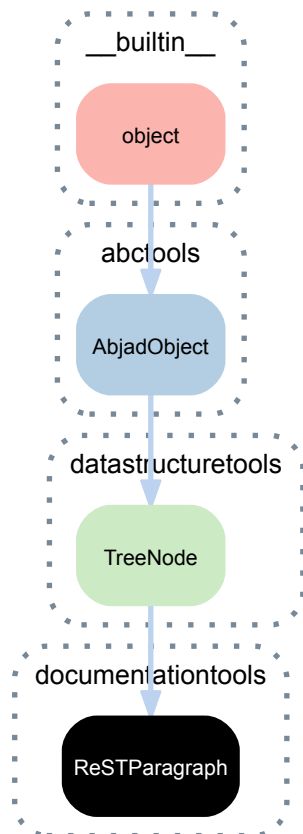
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

(*TreeNode*) .\_\_setstate\_\_(*state*)

### 54.2.22 documentationtools.ReSTParagraph



**class** documentationtools.**ReSTParagraph** (*name=None, text='', wrap=True*)

An ReST paragraph:

```
>>> paragraph = documentationtools.ReSTParagraph(
...     text='blah blah blah')
>>> paragraph
ReSTParagraph(
    text='blah blah blah',
    wrap=True
)
```

```
>>> print _.rest_format
blah blah blah
```

Handles automatic linewrapping.

Return *ReSTParagraph* instance.

#### Bases

- datastructuretools.TreeNode
- abctools.AbjadObject
- \_\_builtin\_\_.object

#### Read-only properties

(TreeNode) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(TreeNode) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

(TreeNode) **.graph\_order**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(*TreeNode*) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

ReSTParagraph **.rest\_format**

(*TreeNode*) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

### Read/write properties

(TreeNode).**name**

ReSTParagraph.**text**

ReSTParagraph.**wrap**

### Special methods

(TreeNode).**\_\_copy\_\_**(\*args)

(TreeNode).**\_\_deepcopy\_\_**(\*args)

(TreeNode).**\_\_eq\_\_**(expr)

(TreeNode).**\_\_getstate\_\_**()

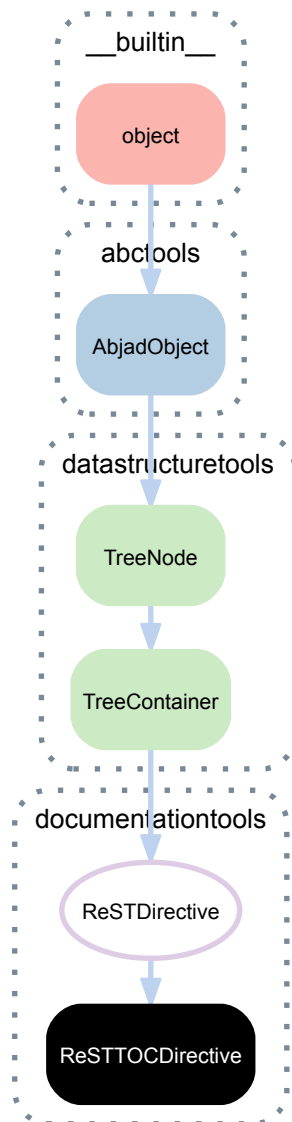
(TreeNode).**\_\_ne\_\_**(expr)

(TreeNode).**\_\_repr\_\_**()

(TreeNode).**\_\_setstate\_\_**(state)



### 54.2.23 documentationtools.ReSTTOCDirective



**class** `documentationtools.ReSTTOCDirective` (*argument=None, children=None, name=None, options=None*)

An ReST TOCTree directive:

```

>>> toc = documentationtools.ReSTTOCDirective()
>>> for item in ['foo/index', 'bar/index', 'baz/index']:
...     toc.append(documentationtools.ReSTTOCItem(text=item))
...
>>> toc.options['maxdepth'] = 1
>>> toc.options['hidden'] = True
>>> toc
ReSTTOCDirective(
  children=(
    ReSTTOCItem(
      text='foo/index'
    ),
    ReSTTOCItem(
      text='bar/index'
    ),
    ReSTTOCItem(
      text='baz/index'
    )
  ),
  options={
    'hidden': True,
  }
)
  
```

```
        'maxdepth': 1
    }
)
```

```
>>> print toc.rest_format
.. toctree::
   :hidden:
   :maxdepth: 1

   foo/index
   bar/index
   baz/index
```

Return *ReSTTOCDirective* instance.

## Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(*TreeContainer*) **.children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Returns tuple of *TreeNode* instances.

(*TreeNode*) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(TreeNode).**depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

ReSTTOCDirective.**directive**

(TreeNode).**graph\_order**

(TreeNode).**improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeContainer).**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
```

```
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Returns tuple.

`ReSTTOCDirective.node_class`

`(TreeContainer).nodes`

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Returns tuple.

`(ReSTDirective).options`

`(TreeNode).parent`

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(*TreeNode*) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

(*ReSTDirective*) **.rest\_format**

(*TreeNode*) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

(*ReSTDirective*) **.argument**

(*TreeNode*) **.name**

## Methods

(*TreeContainer*) **.append** (*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer (
  children=(
    TreeNode(),
  )
)
```

```
>>> a.append(c)
>>> a
TreeContainer (
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.extend** (*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer (
  children=(
    TreeNode(),
    TreeNode()
  )
)
```

Return *None*.

(TreeContainer) **.index** (*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Returns nonnegative integer.

(TreeContainer) **.insert** (*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer (
  children=(
```

```

        TreeNode(),
        TreeNode()
    )
)

```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
    TreeNode()
  )
)

```

Return *None*.

(TreeContainer) **.pop** (*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()

```

```
>>> a.extend([b, c])

```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)

```

```
>>> node = a.pop()

```

```
>>> node == c
True

```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)

```

Returns node.

(TreeContainer) **.remove** (*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()

```

```
>>> a.extend([b, c])

```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode()
  )
)

```

```
>>> a.remove(b)

```

```
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Return *None*.

## Special methods

(TreeContainer).**\_\_contains\_\_**(*expr*)  
 True if *expr* is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Returns boolean.

(TreeNode).**\_\_copy\_\_**(\*args)

(TreeNode).**\_\_deepcopy\_\_**(\*args)

(TreeContainer).**\_\_delitem\_\_**(*i*)  
 Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

(TreeContainer).**\_\_eq\_\_**(*expr*)  
 True if type, duration and children are equivalent, otherwise False.

Returns boolean.

(TreeContainer).**\_\_getitem\_\_**(*i*)  
 Returns node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```



```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

```
(TreeNode).__getstate__()
```

```
(TreeContainer).__iter__()
```

```
(TreeContainer).__len__()
```

Returns nonnegative integer number of nodes in container.

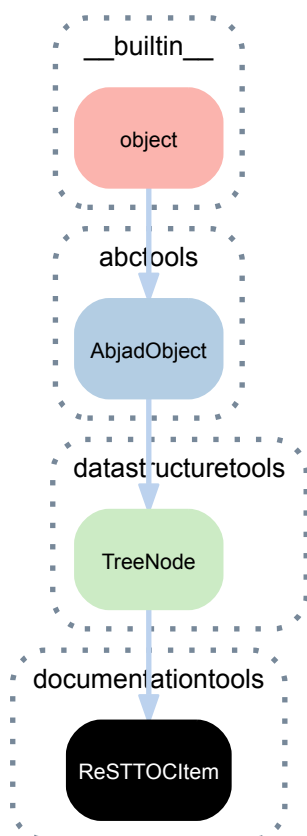
```
(TreeNode).__ne__(expr)
```

```
(TreeNode).__repr__()
```

```
ReSTTOCDirective.__setitem__(i, expr)
```

```
(TreeNode).__setstate__(state)
```

## 54.2.24 documentationtools.ReSTTOCItem



**class** documentationtools.**ReSTTOCItem** (*name=None, text=None*)  
 An ReST TOC item:

```
>>> item = documentationtools.ReSTTOCItem(text='api/index')
>>> item
ReSTTOCItem(
  text='api/index'
)
```

```
>>> print item.rest_format
api/index
```

Return *ReSTTOCItem* instance.

## Bases

- datastructuretools.TreeNode
- abctools.AbjadObject
- \_\_builtin\_\_.object

## Read-only properties

(TreeNode) **.depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(TreeNode) **.depthwise\_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

(TreeNode) **.graph\_order**

(TreeNode) **.improper\_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Returns tuple of *TreeNode* instances.

(TreeNode) **.parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

(TreeNode) **.proper\_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Returns tuple of *TreeNode* instances.

ReSTTOCItem **.rest\_format**

(TreeNode) **.root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

## Read/write properties

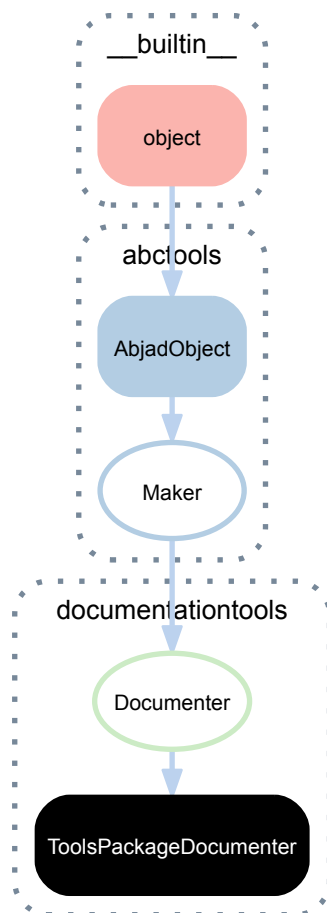
(TreeNode) **.name**

ReSTTOCItem **.text**

## Special methods

```
(TreeNode).__copy__(*args)
(TreeNode).__deepcopy__(*args)
(TreeNode).__eq__(expr)
(TreeNode).__getstate__()
(TreeNode).__ne__(expr)
(TreeNode).__repr__()
(TreeNode).__setstate__(state)
```

### 54.2.25 documentationtools.ToolsPackageDocumenter



**class** `documentationtools.ToolsPackageDocumenter` (*obj*, *ignored\_directory\_names=()*,  
*prefix='abjad.tools.'*)  
 Generates an index for every tools package.

## Bases

- `documentationtools.Documenter`
- `abctools.Maker`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ToolsPackageDocumenter.abstract_class_documenters`

`ToolsPackageDocumenter.all_documenters`

`ToolsPackageDocumenter.concrete_class_documenters`

`ToolsPackageDocumenter.documentation_section`

`ToolsPackageDocumenter.function_documenters`

`ToolsPackageDocumenter.ignored_directory_names`

`ToolsPackageDocumenter.module_name`

`(Documenter).object`

`(Documenter).prefix`

`(Maker).storage_format`

Storage format of maker.

Returns string.

## Methods

`ToolsPackageDocumenter.create_api_toc_section()`

Generate a TOC section to be included in the master API index:

```
>>> module = notetools
>>> documenter = documentationtools.ToolsPackageDocumenter(module)
>>> result = documenter.create_api_toc_section()
```

Returns list.

`(Documenter).new(obj=None, prefix=None)`

## Static methods

`(Documenter).write(file_path, restructured_text)`

## Special methods

`ToolsPackageDocumenter.__call__()`

Generate documentation:

```
>>> module = notetools
>>> documenter = documentationtools.ToolsPackageDocumenter(
...     notetools)
>>> restructured_text = documenter()
```

Returns string.

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
 Interpreter representation of Abjad object.  
 Returns string.

## 54.3 Functions

### 54.3.1 documentationtools.compare\_images

documentationtools.**compare\_images** (*image\_one*, *image\_two*)  
 Compare *image\_one* against *image\_two* using ImageMagick's *compare* commandline tool.  
 Return *True* if images are the same, otherwise *False*.  
 If *compare* is not available, return *False*.

### 54.3.2 documentationtools.make\_ligeti\_example\_lilypond\_file

documentationtools.**make\_ligeti\_example\_lilypond\_file** (*music=None*)  
 Make Ligeti example LilyPond file.  
 Returns LilyPond file.

### 54.3.3 documentationtools.make\_reference\_manual\_graphviz\_graph

documentationtools.**make\_reference\_manual\_graphviz\_graph** (*graph*)  
 Make a GraphvizGraph instance suitable for use in the Abjad reference manual.  
 Returns GraphvizGraph instance.

### 54.3.4 documentationtools.make\_reference\_manual\_lilypond\_file

documentationtools.**make\_reference\_manual\_lilypond\_file** (*music=None*)  
 Make reference manual LilyPond file.

```
>>> score = Score([Staff('c d e f')])
>>> lilypond_file = documentationtools.make_reference_manual_lilypond_file(score)
```

Returns LilyPond file.

### 54.3.5 documentationtools.make\_text\_alignment\_example\_lilypond\_file

documentationtools.**make\_text\_alignment\_example\_lilypond\_file** (*music=None*)  
 Make text-alignment example LilyPond file with *music*.

```
>>> score = Score([Staff('c d e f')])
>>> lilypond_file = documentationtools.make_text_alignment_example_lilypond_file(score)
```

Returns LilyPond file.

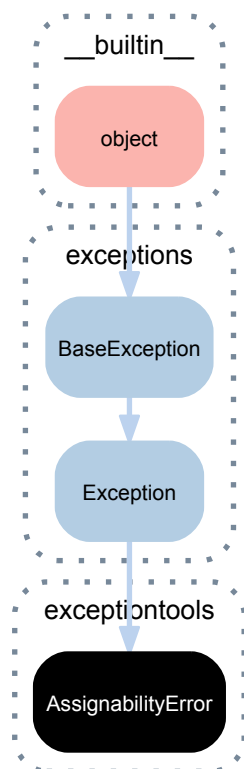




# EXCEPTIONTOOLS

## 55.1 Concrete classes

### 55.1.1 exceptiontools.AssignabilityError



**class** `exceptiontools.AssignabilityError`  
Duration can not be assigned to note, rest or chord.

#### Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

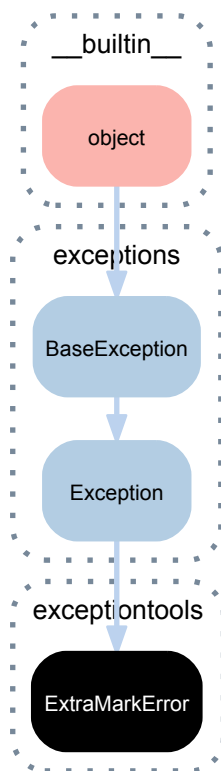
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.2 exceptiontools.ExtraMarkError



**class** `exceptiontools.ExtraMarkError`  
 More than one mark is found for single-mark operation.

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

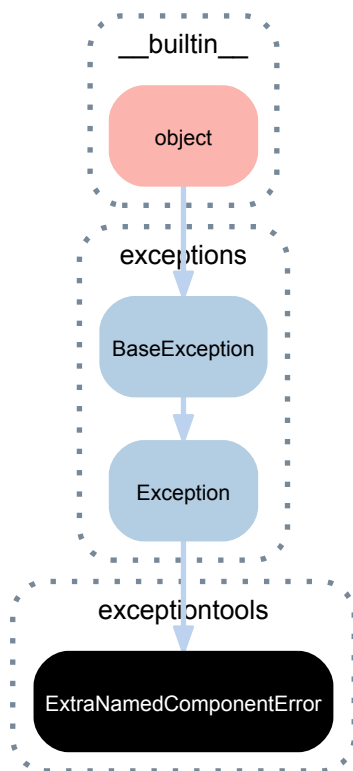
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.3 exceptiontools.ExtraNamedComponentError



**class** exceptiontools.**ExtraNamedComponentError**  
 More than one component with the same name is found..

## Bases

- exceptions.Exception
- exceptions.BaseException
- \_\_builtin\_\_.object

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

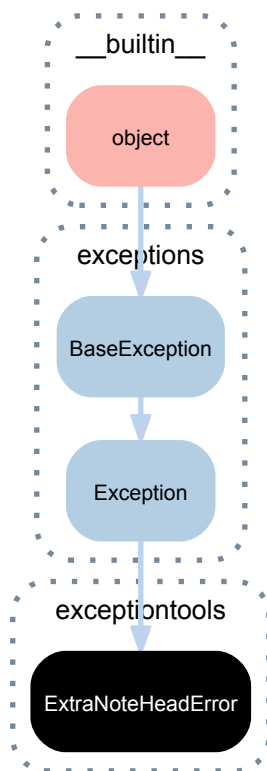
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.4 exceptiontools.ExtraNoteHeadError



**class** `exceptiontools.ExtraNoteHeadError`  
 More than one note head found for single-note head operation.

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

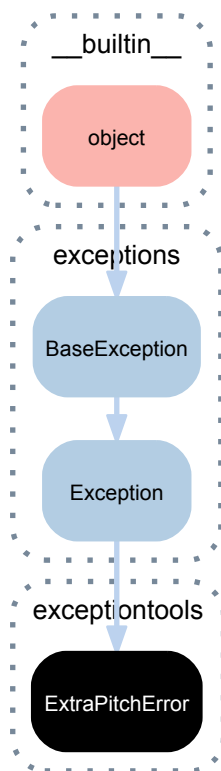
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.5 exceptiontools.ExtraPitchError



**class** exceptiontools.**ExtraPitchError**  
 More than one pitch found for single-pitch operation.

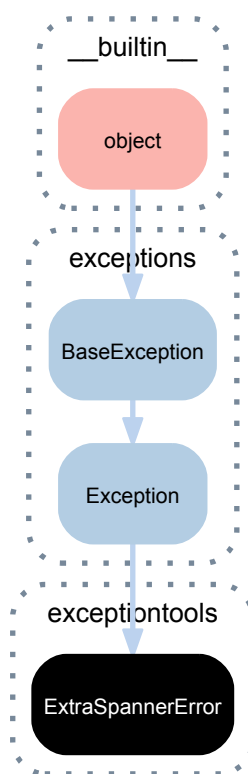
## Bases

- exceptions.Exception
- exceptions.BaseException
- \_\_builtin\_\_.object

## Special methods

```
(BaseException).__delattr__()  
x.__delattr__('name') <==> del x.name  
  
(BaseException).__getitem__()  
x.__getitem__(y) <==> x[y]  
  
(BaseException).__getslice__()  
x.__getslice__(i, j) <==> x[i:j]  
  
Use of negative indices is not supported.  
  
(BaseException).__repr__() <==> repr(x)  
  
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value  
  
(BaseException).__setstate__()  
  
(BaseException).__str__() <==> str(x)  
  
(BaseException).__unicode__()
```

### 55.1.6 exceptiontools.ExtraSpannerError



**class** exceptiontools.**ExtraSpannerError**  
More than one spanner found for single-spanner operation.

## Bases

- exceptions.Exception
- exceptions.BaseException
- \_\_builtin\_\_.object

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

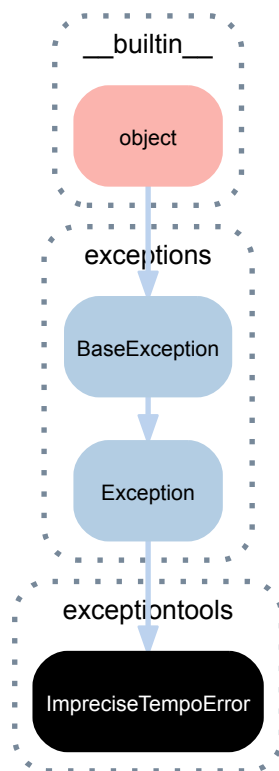
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.7 exceptiontools.ImpreciseTempoError



```
class exceptiontools.ImpreciseTempoError
    TempoMark is imprecise.
```

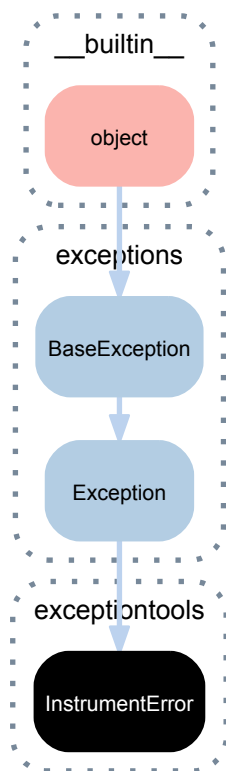
## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()  
x.__delattr__('name') <==> del x.name  
  
(BaseException).__getitem__()  
x.__getitem__(y) <==> x[y]  
  
(BaseException).__getslice__()  
x.__getslice__(i, j) <==> x[i:j]  
  
Use of negative indices is not supported.  
  
(BaseException).__repr__() <==> repr(x)  
  
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value  
  
(BaseException).__setstate__()  
  
(BaseException).__str__() <==> str(x)  
  
(BaseException).__unicode__()
```

### 55.1.8 exceptiontools.InstrumentError



```
class exceptiontools.InstrumentError  
    General instrument error.
```

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`



## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

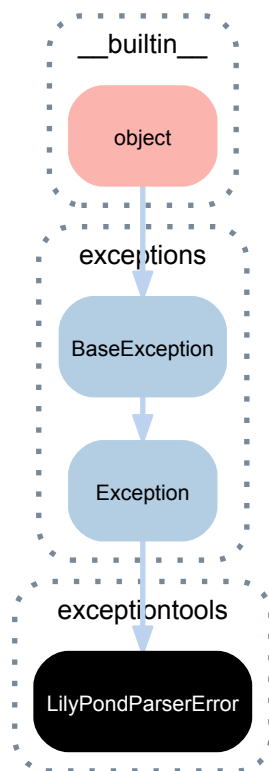
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.9 exceptiontools.LilyPondParserError



```
class exceptiontools.LilyPondParserError
    Can not parse input.
```

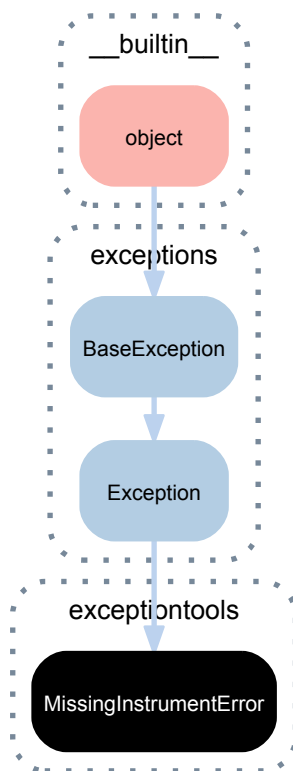
## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()  
x.__delattr__('name') <==> del x.name  
  
(BaseException).__getitem__()  
x.__getitem__(y) <==> x[y]  
  
(BaseException).__getslice__()  
x.__getslice__(i, j) <==> x[i:j]  
  
Use of negative indices is not supported.  
  
(BaseException).__repr__() <==> repr(x)  
  
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value  
  
(BaseException).__setstate__()  
  
(BaseException).__str__() <==> str(x)  
  
(BaseException).__unicode__()
```

### 55.1.10 exceptiontools.MissingInstrumentError



```
class exceptiontools.MissingInstrumentError  
    No instrument found.
```

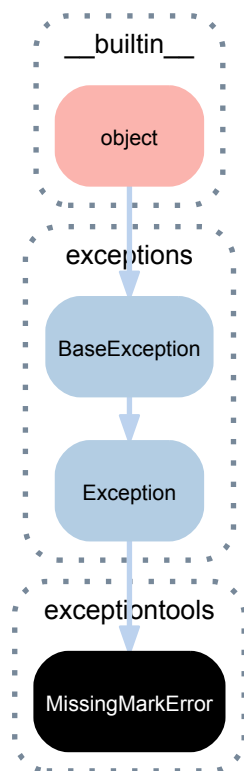
## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()  
x.__delattr__('name') <==> del x.name  
  
(BaseException).__getitem__()  
x.__getitem__(y) <==> x[y]  
  
(BaseException).__getslice__()  
x.__getslice__(i, j) <==> x[i:j]  
  
Use of negative indices is not supported.  
  
(BaseException).__repr__() <==> repr(x)  
  
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value  
  
(BaseException).__setstate__()  
  
(BaseException).__str__() <==> str(x)  
  
(BaseException).__unicode__()
```

### 55.1.11 exceptiontools.MissingMarkError



```
class exceptiontools.MissingMarkError  
    No mark found.
```

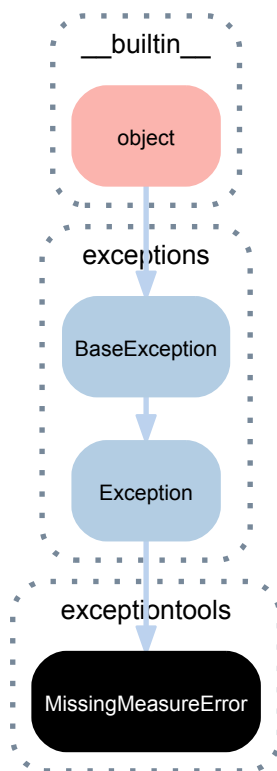
## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()  
x.__delattr__('name') <==> del x.name  
  
(BaseException).__getitem__()  
x.__getitem__(y) <==> x[y]  
  
(BaseException).__getslice__()  
x.__getslice__(i, j) <==> x[i:j]  
  
Use of negative indices is not supported.  
  
(BaseException).__repr__() <==> repr(x)  
  
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value  
  
(BaseException).__setstate__()  
  
(BaseException).__str__() <==> str(x)  
  
(BaseException).__unicode__()
```

### 55.1.12 exceptiontools.MissingMeasureError



```
class exceptiontools.MissingMeasureError  
    No measure found.
```

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

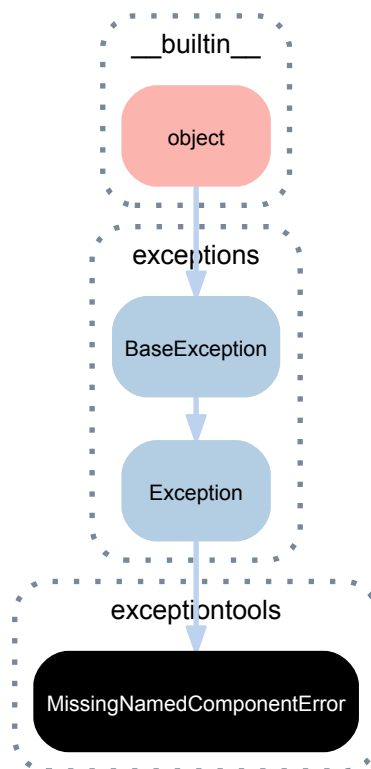
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.13 exceptiontools.MissingNamedComponentError



```
class exceptiontools.MissingNamedComponentError
    No named component found.
```

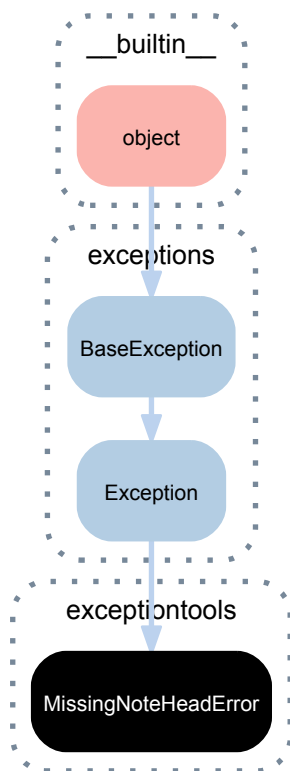
## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()  
x.__delattr__('name') <==> del x.name  
  
(BaseException).__getitem__()  
x.__getitem__(y) <==> x[y]  
  
(BaseException).__getslice__()  
x.__getslice__(i, j) <==> x[i:j]  
  
Use of negative indices is not supported.  
  
(BaseException).__repr__() <==> repr(x)  
  
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value  
  
(BaseException).__setstate__()  
  
(BaseException).__str__() <==> str(x)  
  
(BaseException).__unicode__()
```

### 55.1.14 exceptiontools.MissingNoteHeadError



```
class exceptiontools.MissingNoteHeadError  
    No note head found.
```

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

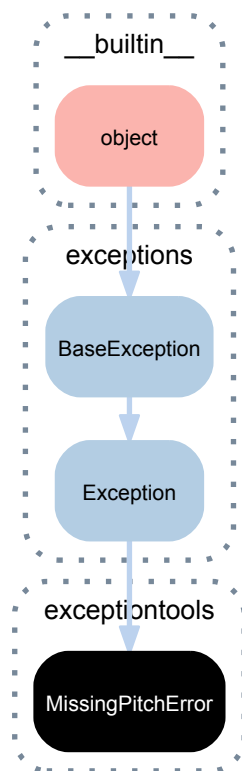
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.15 exceptiontools.MissingPitchError



```
class exceptiontools.MissingPitchError
    No pitch found.
```

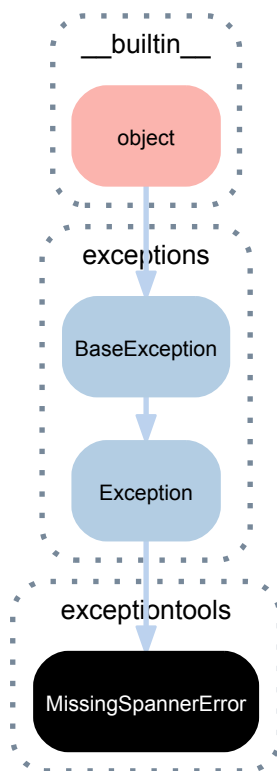
## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()  
    x.__delattr__('name') <==> del x.name  
  
(BaseException).__getitem__()  
    x.__getitem__(y) <==> x[y]  
  
(BaseException).__getslice__()  
    x.__getslice__(i, j) <==> x[i:j]  
  
    Use of negative indices is not supported.  
  
(BaseException).__repr__() <==> repr(x)  
  
(BaseException).__setattr__()  
    x.__setattr__('name', value) <==> x.name = value  
  
(BaseException).__setstate__()  
  
(BaseException).__str__() <==> str(x)  
  
(BaseException).__unicode__()
```

### 55.1.16 exceptiontools.MissingSpannerError



```
class exceptiontools.MissingSpannerError  
    No spanner found.
```

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`



## Special methods

```
(BaseException).__delattr__()
x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

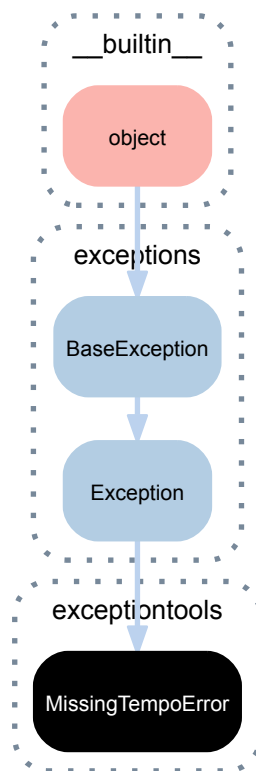
(BaseException).__setattr__()
x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.17 exceptiontools.MissingTempoError



```
class exceptiontools.MissingTempoError
    No tempo found.
```

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

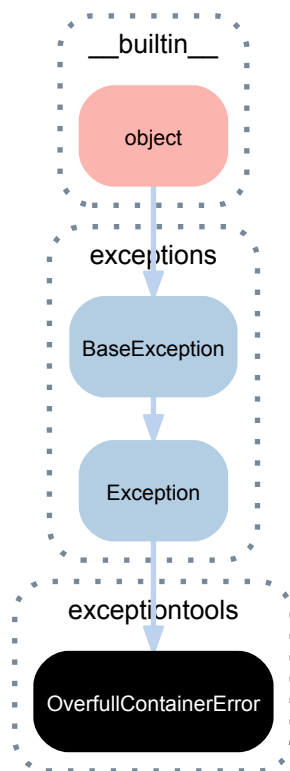
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.18 exceptiontools.OverfullContainerError



**class** `exceptiontools.OverfullContainerError`  
 Container contents duration is greater than container target duration.

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

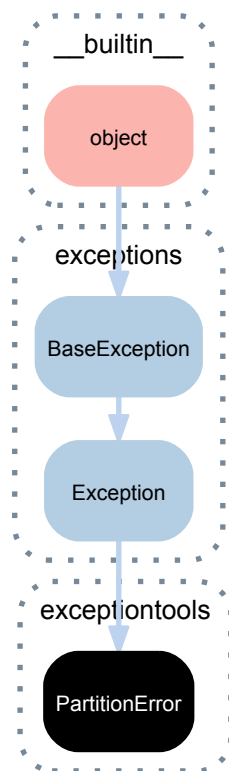
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.19 exceptiontools.PartitionError



**class** `exceptiontools.PartitionError`  
 General partition error.

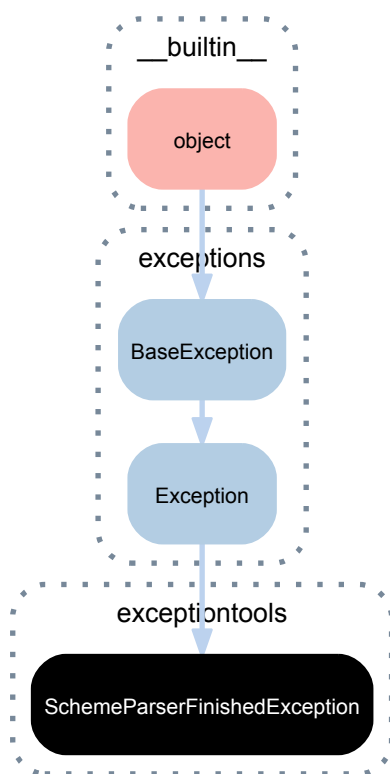
## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()  
    x.__delattr__('name') <==> del x.name  
  
(BaseException).__getitem__()  
    x.__getitem__(y) <==> x[y]  
  
(BaseException).__getslice__()  
    x.__getslice__(i, j) <==> x[i:j]  
  
    Use of negative indices is not supported.  
  
(BaseException).__repr__() <==> repr(x)  
  
(BaseException).__setattr__()  
    x.__setattr__('name', value) <==> x.name = value  
  
(BaseException).__setstate__()  
  
(BaseException).__str__() <==> str(x)  
  
(BaseException).__unicode__()
```

### 55.1.20 exceptiontools.SchemeParserFinishedException



```
class exceptiontools.SchemeParserFinishedException
    SchemeParser has finished parsing.
```

## Bases

- `exceptions.Exception`
- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

```
(BaseException).__delattr__()
    x.__delattr__('name') <==> del x.name

(BaseException).__getitem__()
    x.__getitem__(y) <==> x[y]

(BaseException).__getslice__()
    x.__getslice__(i, j) <==> x[i:j]

    Use of negative indices is not supported.

(BaseException).__repr__() <==> repr(x)

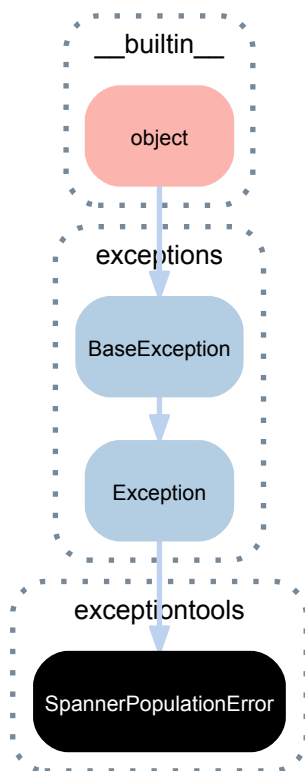
(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__setstate__()

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

### 55.1.21 exceptiontools.SpannerPopulationError



**class** `exceptiontools.SpannerPopulationError`

Spanner contents incorrect.

Spanner may be missing component it is assumed to have.

Spanner may have a component it is assumed not to have.

## Bases

- `exceptions.Exception`

- `exceptions.BaseException`
- `__builtin__.object`

### Special methods

`(BaseException).__delattr__()`  
`x.__delattr__('name') <==> del x.name`

`(BaseException).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(BaseException).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(BaseException).__repr__()` <==> `repr(x)`

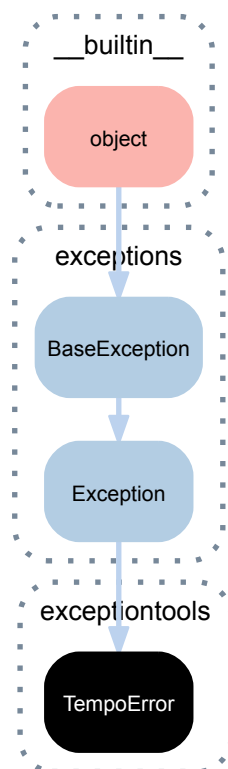
`(BaseException).__setattr__()`  
`x.__setattr__('name', value) <==> x.name = value`

`(BaseException).__setstate__()`

`(BaseException).__str__()` <==> `str(x)`

`(BaseException).__unicode__()`

### 55.1.22 exceptiontools.TempoError



**class** `exceptiontools.TempoError`  
 General tempo error.

### Bases

- `exceptions.Exception`

- `exceptions.BaseException`
- `__builtin__.object`

### Special methods

`(BaseException).__delattr__()`  
`x.__delattr__('name') <==> del x.name`

`(BaseException).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(BaseException).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(BaseException).__repr__()` <==> `repr(x)`

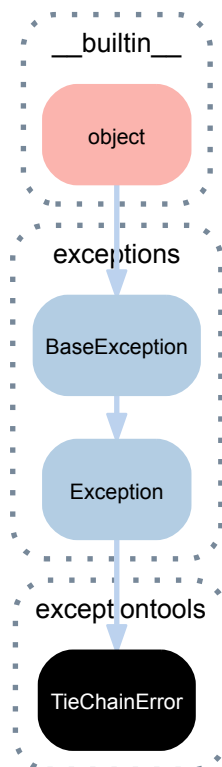
`(BaseException).__setattr__()`  
`x.__setattr__('name', value) <==> x.name = value`

`(BaseException).__setstate__()`

`(BaseException).__str__()` <==> `str(x)`

`(BaseException).__unicode__()`

### 55.1.23 exceptiontools.TieChainError



**class** `exceptiontools.TieChainError`  
 General tie chain error.

### Bases

- `exceptions.Exception`

- `exceptions.BaseException`
- `__builtin__.object`

### Special methods

`(BaseException).__delattr__()`  
`x.__delattr__('name') <==> del x.name`

`(BaseException).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(BaseException).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(BaseException).__repr__()` <==> `repr(x)`

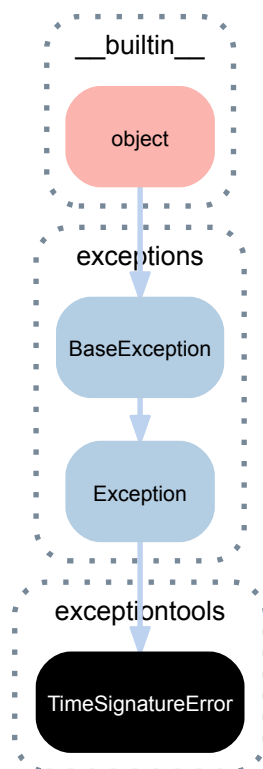
`(BaseException).__setattr__()`  
`x.__setattr__('name', value) <==> x.name = value`

`(BaseException).__setstate__()`

`(BaseException).__str__()` <==> `str(x)`

`(BaseException).__unicode__()`

### 55.1.24 exceptiontools.TimeSignatureError



**class** `exceptiontools.TimeSignatureError`  
 General time signature error.

### Bases

- `exceptions.Exception`



- `exceptions.BaseException`
- `__builtin__.object`

### Special methods

`(BaseException).__delattr__()`  
`x.__delattr__('name') <==> del x.name`

`(BaseException).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(BaseException).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(BaseException).__repr__()` <==> `repr(x)`

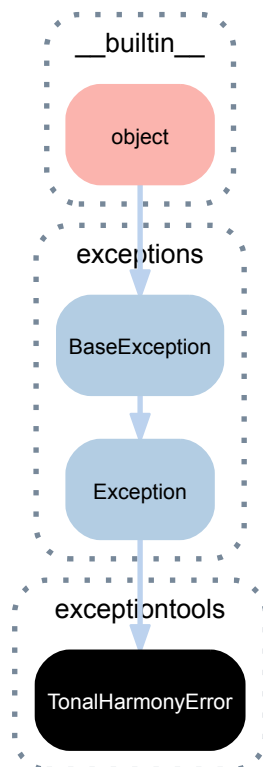
`(BaseException).__setattr__()`  
`x.__setattr__('name', value) <==> x.name = value`

`(BaseException).__setstate__()`

`(BaseException).__str__()` <==> `str(x)`

`(BaseException).__unicode__()`

### 55.1.25 exceptiontools.TonalHarmonyError



**class** `exceptiontools.TonalHarmonyError`  
 General tonal harmony error.

### Bases

- `exceptions.Exception`

- `exceptions.BaseException`
- `__builtin__.object`

## Special methods

`(BaseException).__delattr__()`  
`x.__delattr__('name') <==> del x.name`

`(BaseException).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(BaseException).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(BaseException).__repr__()` <==> `repr(x)`

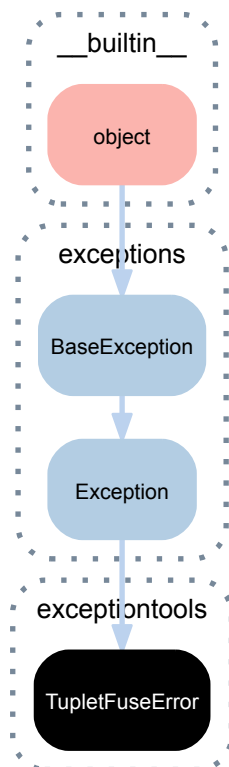
`(BaseException).__setattr__()`  
`x.__setattr__('name', value) <==> x.name = value`

`(BaseException).__setstate__()`

`(BaseException).__str__()` <==> `str(x)`

`(BaseException).__unicode__()`

## 55.1.26 exceptiontools.TupletFuseError



**class** `exceptiontools.TupletFuseError`

Tuplets must carry same multiplier and be same type in order to fuse correctly.

## Bases

- `exceptions.Exception`

- `exceptions.BaseException`
- `__builtin__.object`

### Special methods

`(BaseException).__delattr__()`  
`x.__delattr__('name') <==> del x.name`

`(BaseException).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(BaseException).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(BaseException).__repr__()` <==> `repr(x)`

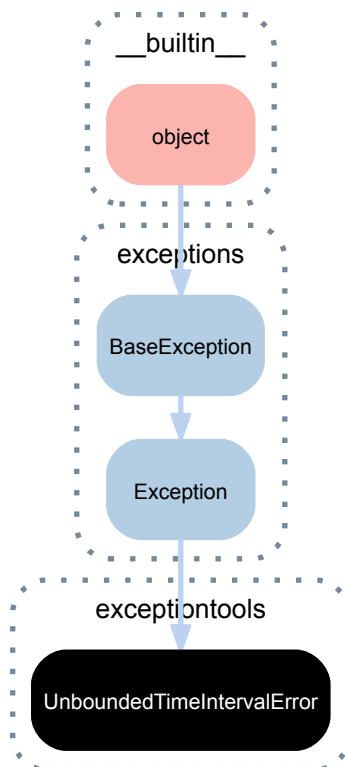
`(BaseException).__setattr__()`  
`x.__setattr__('name', value) <==> x.name = value`

`(BaseException).__setstate__()`

`(BaseException).__str__()` <==> `str(x)`

`(BaseException).__unicode__()`

### 55.1.27 `exceptiontools.UnboundedTimeIntervalError`



**class** `exceptiontools.UnboundedTimeIntervalError`  
 Time interval has no bounds.

### Bases

- `exceptions.Exception`

- `exceptions.BaseException`
- `__builtin__.object`

### Special methods

`(BaseException).__delattr__()`  
`x.__delattr__('name') <==> del x.name`

`(BaseException).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(BaseException).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(BaseException).__repr__()` <==> `repr(x)`

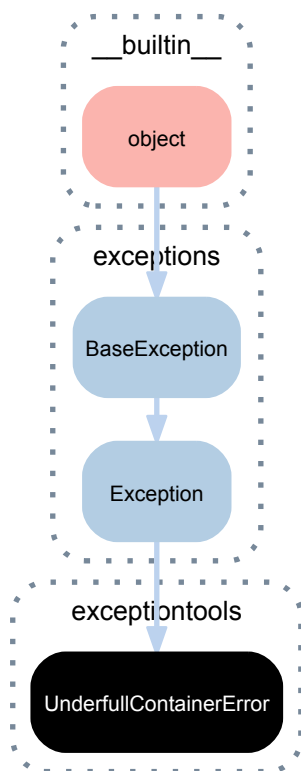
`(BaseException).__setattr__()`  
`x.__setattr__('name', value) <==> x.name = value`

`(BaseException).__setstate__()`

`(BaseException).__str__()` <==> `str(x)`

`(BaseException).__unicode__()`

### 55.1.28 `exceptiontools.UnderfullContainerError`



**class** `exceptiontools.UnderfullContainerError`

Container contents duration is less than container target duration.

### Bases

- `exceptions.Exception`

- `exceptions.BaseException`
- `__builtin__.object`

### Special methods

`(BaseException).__delattr__()`  
`x.__delattr__('name') <==> del x.name`

`(BaseException).__getitem__()`  
`x.__getitem__(y) <==> x[y]`

`(BaseException).__getslice__()`  
`x.__getslice__(i, j) <==> x[i:j]`

Use of negative indices is not supported.

`(BaseException).__repr__()` <==> `repr(x)`

`(BaseException).__setattr__()`  
`x.__setattr__('name', value) <==> x.name = value`

`(BaseException).__setstate__()`

`(BaseException).__str__()` <==> `str(x)`

`(BaseException).__unicode__()`



# FORMATTOOLS

## 56.1 Functions

### 56.1.1 `formattools.format_lilypond_attribute`

`formattools.format_lilypond_attribute` (*attribute*)  
Formats LilyPond attribute according to Scheme formatting conventions.  
Returns string.

### 56.1.2 `formattools.format_lilypond_value`

`formattools.format_lilypond_value` (*value*)  
Formats LilyPond value according to Scheme formatting conventions.  
Returns string.

### 56.1.3 `formattools.get_all_format_contributions`

`formattools.get_all_format_contributions` (*component*)  
Get all format contributions for *component*.  
Returns nested dictionary.

### 56.1.4 `formattools.get_all_mark_format_contributions`

`formattools.get_all_mark_format_contributions` (*component*)  
Get all mark format contributions as nested dictionaries.  
The first level of keys represent format slots.  
The second level of keys represent format contributor ('articulations', 'markup', etc.).  
Returns dict.

### 56.1.5 `formattools.get_articulation_format_contributions`

`formattools.get_articulation_format_contributions` (*component*)  
Get articulation format contributions for *component*.  
Returns list.

### 56.1.6 `formattools.get_comment_format_contributions_for_slot`

`formattools.get_comment_format_contributions_for_slot` (*component*, *slot*)  
Get comment format contributions for *component* at *slot*.  
Returns list.

### 56.1.7 `formattools.get_context_mark_format_contributions_for_slot`

`formattools.get_context_mark_format_contributions_for_slot` (*component*, *slot*)  
Get context mark format contributions for *component* at *slot*.  
Returns list.

### 56.1.8 `formattools.get_context_mark_format_pieces`

`formattools.get_context_mark_format_pieces` (*mark*)  
Get context mark format pieces for *mark*.  
Returns list.

### 56.1.9 `formattools.get_context_setting_format_contributions`

`formattools.get_context_setting_format_contributions` (*component*)  
Get context setting format contributions for *component*.  
Returns sorted list.

### 56.1.10 `formattools.get_grob_override_format_contributions`

`formattools.get_grob_override_format_contributions` (*component*)  
Get grob override format contributions for *component*.  
Returns alphabetized list of LilyPond grob overrides.

### 56.1.11 `formattools.get_grob_revert_format_contributions`

`formattools.get_grob_revert_format_contributions` (*component*)  
Get grob revert format contributions.  
Returns alphabetized list of LilyPond grob reverts.

### 56.1.12 `formattools.get_lilypond_command_mark_format_contributions_for_slot`

`formattools.get_lilypond_command_mark_format_contributions_for_slot` (*component*,  
*slot*)  
Get LilyPond command mark format contributions for *component* at *slot*.  
Returns list.

### 56.1.13 `formattools.get_markup_format_contributions`

`formattools.get_markup_format_contributions` (*component*)  
Get markup format contributions for *component*.  
Returns list.



### 56.1.14 `formattools.get_spanner_format_contributions`

`formattools.get_spanner_format_contributions` (*component*)

Gets spanner format contributions for *component*.

Dictionary keys equal to format slot; dictionary values equal to format contributions.

### 56.1.15 `formattools.get_stem_tremolo_format_contributions`

`formattools.get_stem_tremolo_format_contributions` (*component*)

Get stem tremolo format contributions for *component*.

Returns list.

### 56.1.16 `formattools.is_formattable_context_mark_for_component`

`formattools.is_formattable_context_mark_for_component` (*mark*, *component*)

Returns True if ContextMark *mark* can format for *component*.

### 56.1.17 `formattools.make_lilypond_override_string`

`formattools.make_lilypond_override_string` (*grob\_name*, *grob\_attribute*, *grob\_value*,  
*context\_name=None*, *is\_once=False*)

Makes Lilypond override string.

Does not include once indicator.

Returns string.

### 56.1.18 `formattools.make_lilypond_revert_string`

`formattools.make_lilypond_revert_string` (*grob\_name*, *grob\_attribute*, *con-*  
*text\_name=None*)

Makes LilyPond revert string.

Returns string.

### 56.1.19 `formattools.report_component_format_contributions`

`formattools.report_component_format_contributions` (*component*, *verbose=False*)

Report *component* format contributions:

```
>>> staff = Staff("c'4 [ ( d'4 e'4 f'4 ] )")
>>> staff[0].override.note_head.color = 'red'
```

```
>>> print formattools.report_component_format_contributions(staff[0])
slot 1:
  grob overrides:
    \once \override NoteHead #'color = #red
slot 3:
slot 4:
  leaf body:
    c'4 [ (
slot 5:
slot 7:
```

Returns string.

### 56.1.20 `formattools.report_spanner_format_contributions`

`formattools.report_spanner_format_contributions` (*spanner*)

Report spanner format contributions for every leaf to which spanner attaches:

```
>>> staff = Staff("c8 d e f")
>>> spanner = spannertools.BeamSpanner(staff[:])

>>> print formattools.report_spanner_format_contributions(spanner)
c8 before: []
   after: []
   right: ['']

d8 before: []
   after: []
   right: []

e8 before: []
   after: []
   right: []

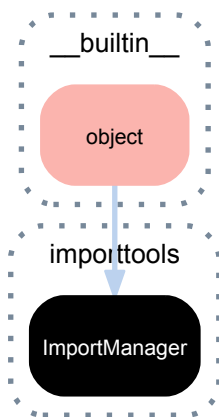
f8 before: []
   after: []
   right: ['']
```

Returns none or return string.

# IMPORTTOOLS

## 57.1 Concrete classes

### 57.1.1 importtools.ImportManager



**class** `importtools.ImportManager`  
Imports structured packages.

#### Bases

- `__builtin__.object`

#### Static methods

`ImportManager.import_public_names_from_filesystem_path_into_namespace` (*path*,  
*namespace*,  
*package\_root\_name*='abjad')

Inspect the top level of *path*.

Find .py modules in *path* and import public functions from .py modules into *namespace*.

Find packages in *path* and import package names into *namespace*.

Do not import package content into *namespace*.

Do not inspect lower levels of *path*.

`ImportManager.import_structured_package` (*path*, *namespace*, *package\_root\_name*='abjad')

Import public names from *path* into *namespace*.

This is the custom function that all Abjad packages use to import public classes and functions on startup.

The function will work for any package laid out like Abjad packages.

Set *package\_root\_name* to the root any Abjad-like package structure.

Returns none.

# INTROSPECTIONTOOLS

## 58.1 Functions

### 58.1.1 introspectiontools.class\_to\_tools\_package\_qualified\_class\_name

`introspectiontools.class_to_tools_package_qualified_class_name(current_class)`  
Change *current\_class* to tools package-qualified class name:

```
>>> introspectiontools.class_to_tools_package_qualified_class_name(Note)
'notetools.Note'
```

Returns string.

### 58.1.2 introspectiontools.get\_current\_function\_name

`introspectiontools.get_current_function_name()`  
Get current function name:

```
>>> def foo():
...     function_name = introspectiontools.get_current_function_name()
...     print 'Function name is {!r}'.format(function_name)
```

```
>>> foo()
Function name is 'foo'.
```

Call this function within the implementation of any ofther function.

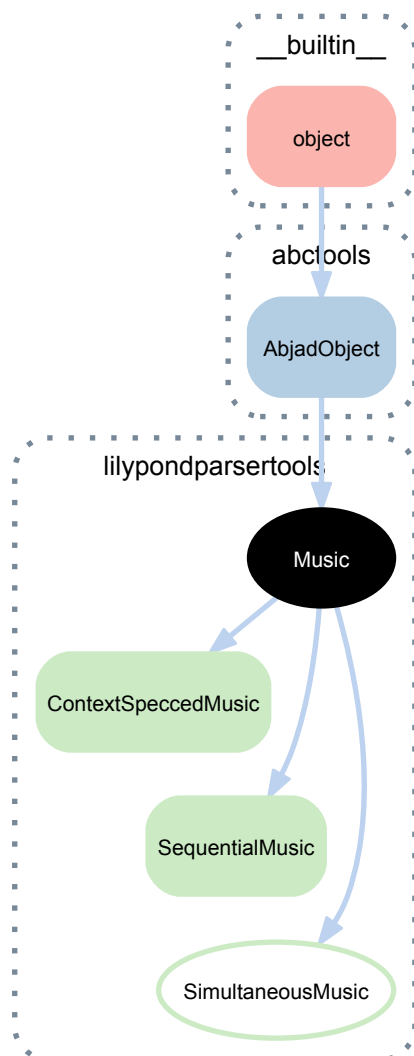
Returns enclosing function name as a string or else none.



# LILYPONDPARSERTOOLS

## 59.1 Abstract classes

### 59.1.1 lilypondparsertools.Music



```
class lilypondparsertools.Music(music)
    Abjad model of the LilyPond AST music node.
```

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

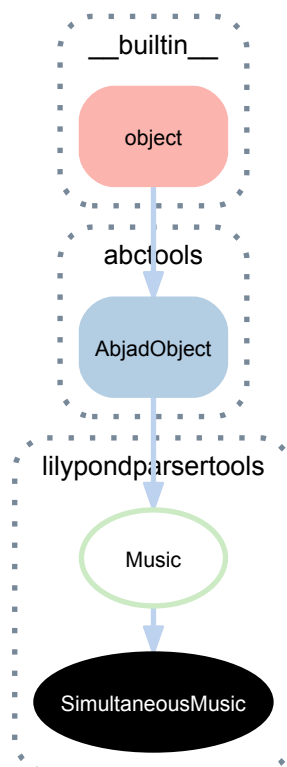
## Methods

`Music.construct()`

## Special methods

- `(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.
- `(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.
- `(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 59.1.2 lilypondparsertools.SimultaneousMusic



**class** `lilypondparsertools.SimultaneousMusic` (*music*)  
 Abjad model of the LilyPond AST simultaneous music node.



## Bases

- `lilypondparsertools.Music`
- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

`(Music).construct()`

## Special methods

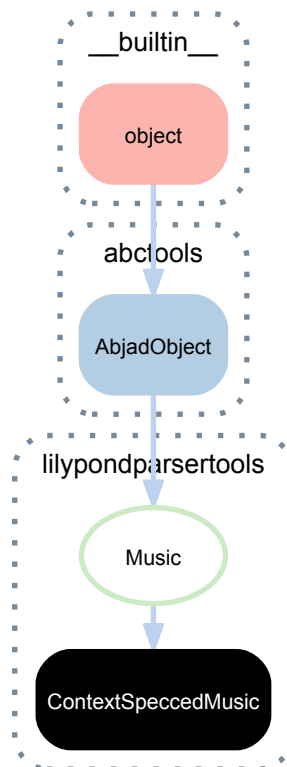
`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

## 59.2 Concrete classes

### 59.2.1 lilypondparsertools.ContextSpeccedMusic



**class** `lilypondparsertools.ContextSpeccedMusic` (*context*, *optional\_id*, *optional\_context\_mod*, *music*) *op-*  
 Abjad model of the LilyPond AST context-specced music node.

## Bases

- `lilypondparsertools.Music`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`ContextSpeccedMusic.known_contexts`

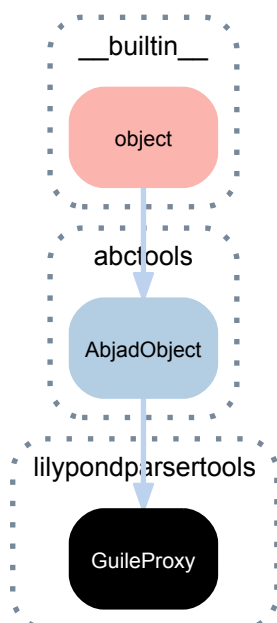
## Methods

`ContextSpeccedMusic.construct()`

## Special methods

- (`AbjadObject`).`__eq__`(*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.
- (`AbjadObject`).`__ne__`(*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.
- (`AbjadObject`).`__repr__`()  
 Interpreter representation of Abjad object.  
 Returns string.

## 59.2.2 lilypondparsertools.GuileProxy



**class** `lilypondparsertools.GuileProxy` (*client*)  
 Emulates LilyPond music functions.  
 Used internally by LilyPondParser.  
 Not composer-safe.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

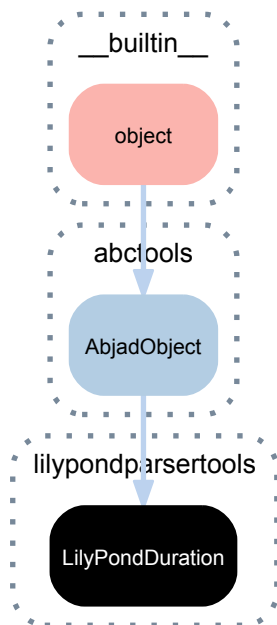
`GuileProxy.acciaccatura` (*music*)  
`GuileProxy.appoggiatura` (*music*)  
`GuileProxy.bar` (*string*)  
`GuileProxy.breathe` ()  
`GuileProxy.clef` (*string*)  
`GuileProxy.grace` (*music*)  
`GuileProxy.key` (*notename\_pitch*, *number\_list*)  
`GuileProxy.language` (*string*)  
`GuileProxy.makeClusters` (*music*)  
`GuileProxy.mark` (*label*)  
`GuileProxy.one_voice` ()  
`GuileProxy.relative` (*pitch*, *music*)  
`GuileProxy.skip` (*duration*)  
`GuileProxy.slashed_grace_container` (*music*)  
`GuileProxy.time` (*number\_list*, *fraction*)  
`GuileProxy.times` (*fraction*, *music*)  
`GuileProxy.transpose` (*from\_pitch*, *to\_pitch*, *music*)  
`GuileProxy.voiceFour` ()  
`GuileProxy.voiceOne` ()  
`GuileProxy.voiceThree` ()  
`GuileProxy.voiceTwo` ()

## Special methods

`GuileProxy.__call__` (*function\_name*, *args*)  
`(AbjadObject).__eq__` (*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.  
`(AbjadObject).__ne__` (*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 59.2.3 lilypondparsertools.LilyPondDuration



**class** `lilypondparsertools.LilyPondDuration` (*duration*, *multiplier=None*)  
 Model of a duration in LilyPond.  
 Not composer-safe.  
 Used internally by LilyPondParser.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

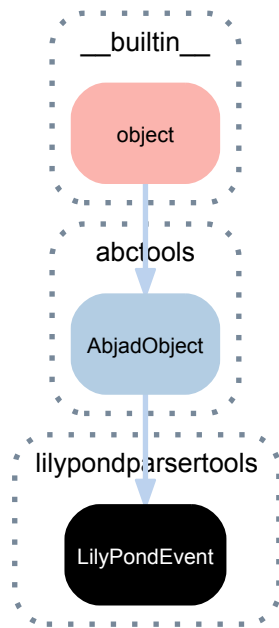
#### Special methods

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 59.2.4 lilypondparsertools.LilyPondEvent



**class** `lilypondparsertools.LilyPondEvent` (*name*, *\*\*kwargs*)  
 Model of an arbitrary event in LilyPond.  
 Not composer-safe.  
 Used internally by LilyPondParser.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

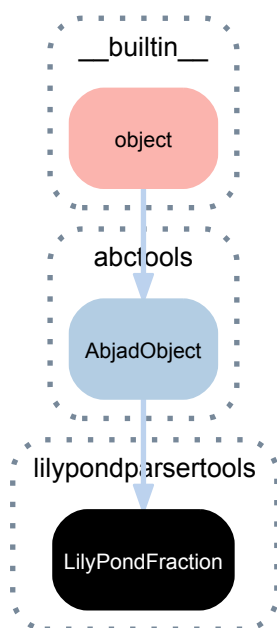
#### Special methods

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`LilyPondEvent.__repr__()`

### 59.2.5 lilypondparsertools.LilyPondFraction



**class** `lilypondparsertools.LilyPondFraction` (*numerator, denominator*)

Model of a fraction in LilyPond.

Not composer-safe.

Used internally by LilyPondParser.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`(AbjadObject).__eq__(expr)`

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

`(AbjadObject).__ne__(expr)`

True when ID of *expr* does not equal ID of Abjad object.

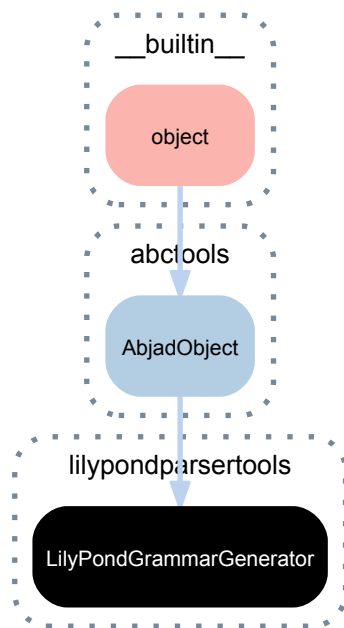
Returns boolean.

`(AbjadObject).__repr__()`

Interpreter representation of Abjad object.

Returns string.

### 59.2.6 lilypondparsertools.LilyPondGrammarGenerator



**class** lilypondparsertools.LilyPondGrammarGenerator  
Generate a syntax skeleton from LilyPond grammar files.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

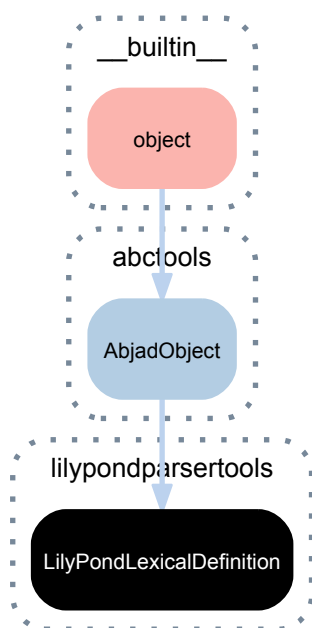
`LilyPondGrammarGenerator.__call__(skeleton_path, parser_output_path, parser_tab_hh_path)`

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`(AbjadObject).__repr__()`  
Interpreter representation of Abjad object.  
Returns string.

### 59.2.7 lilypondparsertools.LilyPondLexicalDefinition



**class** `lilypondparsertools.LilyPondLexicalDefinition` (*client*)  
The lexical definition of LilyPond’s syntax.  
Effectively equivalent to LilyPond’s `lexer.ll` file.  
Not composer-safe.  
Used internally by `LilyPondParser`.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Methods

`LilyPondLexicalDefinition.push_signature` (*signature*, *t*)  
`LilyPondLexicalDefinition.scan_bare_word` (*t*)  
`LilyPondLexicalDefinition.scan_escaped_word` (*t*)  
`LilyPondLexicalDefinition.t_651_a` (*t*)  
    `(([-?[0-9]+).[0-9]*)([-?[0-9]+)`  
`LilyPondLexicalDefinition.t_651_b` (*t*)  
    `-[0-9]+`  
`LilyPondLexicalDefinition.t_661` (*t*)  
    `-.`  
`LilyPondLexicalDefinition.t_666` (*t*)  
    `[0-9]+`  
`LilyPondLexicalDefinition.t_ANY_165` (*t*)  
    `r`  
`LilyPondLexicalDefinition.t_INITIAL_643` (*t*)  
    `[a-zA-Z200-377](((a-zA-Z200-377)|_)[0-9])|-*`



```

LilyPondLexicalDefinition.t_INITIAL_646 (t)
  \[a-zA-Z200-377](((a-zA-Z200-377)|_)[0-9])|~)*

LilyPondLexicalDefinition.t_INITIAL_markup_notes_210 (t)
  %{

LilyPondLexicalDefinition.t_INITIAL_markup_notes_214 (t)
  %[^{nr}[^nr]*nr]

LilyPondLexicalDefinition.t_INITIAL_markup_notes_216 (t)
  %[^{nr]

LilyPondLexicalDefinition.t_INITIAL_markup_notes_218 (t)
  %nr]

LilyPondLexicalDefinition.t_INITIAL_markup_notes_220 (t)
  %[^{nr}[^nr]*

LilyPondLexicalDefinition.t_INITIAL_markup_notes_222 (t)
  [

  ]

LilyPondLexicalDefinition.t_INITIAL_markup_notes_227 (t)
  “

LilyPondLexicalDefinition.t_INITIAL_markup_notes_353 (t)
  #

LilyPondLexicalDefinition.t_INITIAL_notes_233 (t)
  \version[ ntrf]*

LilyPondLexicalDefinition.t_INITIAL_notes_387 (t)
  <<

LilyPondLexicalDefinition.t_INITIAL_notes_390 (t)
  >>

LilyPondLexicalDefinition.t_INITIAL_notes_396 (t)
  <

LilyPondLexicalDefinition.t_INITIAL_notes_399 (t)
  >

LilyPondLexicalDefinition.t_INITIAL_notes_686 (t)
  \.

LilyPondLexicalDefinition.t_error (t)

LilyPondLexicalDefinition.t_longcomment_291 (t)
  [^%]+

LilyPondLexicalDefinition.t_longcomment_293 (t)
  %+[^}%]*

LilyPondLexicalDefinition.t_longcomment_296 (t)
  %}

LilyPondLexicalDefinition.t_longcomment_error (t)

LilyPondLexicalDefinition.t_markup_545 (t)
  \score

LilyPondLexicalDefinition.t_markup_548 (t)
  \([a-zA-Z200-377]|[-_])~)+

LilyPondLexicalDefinition.t_markup_601 (t)
  [^#{ }~\ ntrf]+

LilyPondLexicalDefinition.t_markup_error (t)

```

```

LilyPondLexicalDefinition.t_newline (t)
    n+
LilyPondLexicalDefinition.t_notes_417 (t)
    [a-zA-Z200-377]+
LilyPondLexicalDefinition.t_notes_421 (t)
    \[a-zA-Z200-377]+
LilyPondLexicalDefinition.t_notes_424 (t)
    [0-9]+/[0-9]+
LilyPondLexicalDefinition.t_notes_428 (t)
    [0-9]+//
LilyPondLexicalDefinition.t_notes_428b (t)
    [0-9]+
LilyPondLexicalDefinition.t_notes_433 (t)
    \[0-9]+
LilyPondLexicalDefinition.t_notes_error (t)
LilyPondLexicalDefinition.t_quote_440 (t)
    [nt\''"]
LilyPondLexicalDefinition.t_quote_443 (t)
    [^\\""]+
LilyPondLexicalDefinition.t_quote_446 (t)
    “
LilyPondLexicalDefinition.t_quote_456 (t)
    .
LilyPondLexicalDefinition.t_quote_XXX (t)
    \”
LilyPondLexicalDefinition.t_quote_error (t)
LilyPondLexicalDefinition.t_scheme_error (t)
LilyPondLexicalDefinition.t_version_242 (t)
    “[^”]*”
LilyPondLexicalDefinition.t_version_278 (t)
    (.ln)
LilyPondLexicalDefinition.t_version_341 (t)
    “[^”]*
LilyPondLexicalDefinition.t_version_error (t)

```

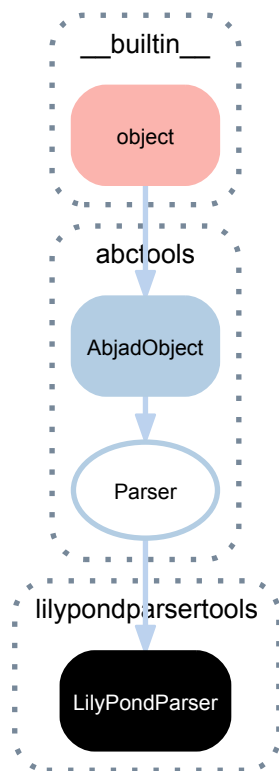
## Special methods

```

(AbjadObject) .__eq__ (expr)
    True when ID of expr equals ID of Abjad object.
    Returns boolean.
(AbjadObject) .__ne__ (expr)
    True when ID of expr does not equal ID of Abjad object.
    Returns boolean.
(AbjadObject) .__repr__ ()
    Interpreter representation of Abjad object.
    Returns string.

```

### 59.2.8 lilypondparsertools.LilyPondParser



**class** `lilypondparsertools.LilyPondParser` (*default\_language='english', debug=False*)  
 Parses a subset of LilyPond input syntax:

```

>>> parser = lilypondparsertools.LilyPondParser()
>>> input = r"\new Staff { c'4 ( d'8 e' fs'2) \fermata }"
>>> result = parser(input)
>>> f(result)
\new Staff {
  c'4 (
    d'8
    e'8
    fs'2 -\fermata )
}
  
```

`LilyPondParser` defaults to English note names, but any of the other languages supported by LilyPond may be used:

```

>>> parser = lilypondparsertools.LilyPondParser('nederlands')
>>> input = '{ c des e fis }'
>>> result = parser(input)
>>> f(result)
{
  c4
  df4
  e4
  fs4
}
  
```

Briefly, `LilyPondParser` understands theses aspects of LilyPond syntax:

- Notes, chords, rests, skips and multi-measure rests
- Durations, dots, and multipliers
- All pitchnames, and octave ticks
- Simple markup (i.e. `c'4 ^ "hello!"`)
- Most articulations

- Most spanners, including beams, slurs, phrasing slurs, ties, and glissandi
- Most context types via `\new` and `\context`, as well as context ids (i.e. `\new Staff = "foo" { }`)
- Variable assignment (i.e. `global = { \time 3/4 } \new Staff { \global }`)
- Many music functions:**
  - `\acciaccatura`
  - `\appoggiatura`
  - `\bar`
  - `\breathe`
  - `\clef`
  - `\grace`
  - `\key`
  - `\transpose`
  - `\language`
  - `\makeClusters`
  - `\mark`
  - `\oneVoice`
  - `\relative`
  - `\skip`
  - `\slashedGrace`
  - `\time`
  - `\times`
  - `\transpose`
  - `\voiceOne`, `\voiceTwo`, `\voiceThree`, `\voiceFour`

LilyPondParser currently **DOES NOT** understand many other aspects of LilyPond syntax:

- `\markup`
- `\book`, `\bookpart`, `\header`, `\layout`, `\midi` and `\paper`
- `\repeat` and `\alternative`
- Lyrics
- `\chordmode`, `\drummode` or `\figuremode`
- Property operations, such as `\override`, `\revert`, `\set`, `\unset`, and `\once`
- Music functions which generate or extensively mutate musical structures
- Embedded Scheme statements (anything beginning with `#`)

Returns LilyPondParser instance.

## Bases

- `abctools.Parser`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

`LilyPondParser.available_languages`

Tuple of pitch-name languages supported by `LilyPondParser`:

```
>>> parser = lilypondparsertools.LilyPondParser()
>>> for language in parser.available_languages:
...     print language
catalan
deutsch
english
espanol
español
français
italiano
nederlands
norsk
portugues
suomi
svenska
vlaams
```

Returns tuple.

`(Parser).debug`

True if the parser runs in debugging mode.

`(Parser).lexer`

The parser's PLY Lexer instance.

`LilyPondParser.lexer_rules_object`

`(Parser).logger`

The parser's Logger instance.

`(Parser).logger_path`

The output path for the parser's logfile.

`(Parser).output_path`

The output path for files associated with the parser.

`(Parser).parser`

The parser's PLY LRPParser instance.

`LilyPondParser.parser_rules_object`

`(Parser).pickle_path`

The output path for the parser's pickled parsing tables.

## Read/write properties

`LilyPondParser.default_language`

Read/write attribute to set parser's default pitch-name language:

```
>>> parser = lilypondparsertools.LilyPondParser()
```

```
>>> parser.default_language
'english'
```

```
>>> parser('{ c df e fs }')
{c4, df4, e4, fs4}
```

```
>>> parser.default_language = 'nederlands'
>>> parser.default_language
'nederlands'
```

```
>>> parser('{ c des e fis }')
{c4, df4, e4, fs4}
```

Returns string.

## Methods

(Parser).**.tokenize**(*input\_string*)  
Tokenize *input string* and print results.

## Class methods

LilyPondParser.**.register\_markup\_function**(*name*, *signature*)

Register a custom markup function globally with LilyPondParser:

```
>>> name = 'my-custom-markup-function'
>>> signature = ['markup?']
>>> lilypondparsertools.LilyPondParser.register_markup_function(name, signature)
```

```
>>> parser = lilypondparsertools.LilyPondParser()
>>> string = r"\markup { \my-custom-markup-function { foo bar baz } }"
>>> parser(string)
Markup( (MarkupCommand('my-custom-markup-function', ['foo', 'bar', 'baz']),))
```

*signature* should be a sequence of zero or more type-predicate names, as understood by LilyPond. Consult LilyPond's documentation for a complete list of all understood type-predicates.

Returns None

## Static methods

LilyPondParser.**.list\_known\_contexts**()

List all LilyPond contexts recognized by LilyPondParser:

```
>>> for x in lilypondparsertools.LilyPondParser.list_known_contexts():
...     print x
...
ChoirStaff
ChordNames
CueVoice
Devnul1
DrumStaff
DrumVoice
Dynamics
FiguredBass
FretBoards
Global
GrandStaff
GregorianTranscriptionStaff
GregorianTranscriptionVoice
KievanStaff
KievanVoice
Lyrics
MensuralStaff
MensuralVoice
NoteNames
PetrucciStaff
PetrucciVoice
PianoStaff
RhythmicStaff
Score
Staff
StaffGroup
TabStaff
TabVoice
```

VaticanaStaff  
 VaticanaVoice  
 Voice

Returns list.

`LilyPondParser.list_known_grobs()`

List all LilyPond grobs recognized by LilyPondParser:

```
>>> for x in lilypondparsertools.LilyPondParser.list_known_grobs():
...     print x
...
Accidental
AccidentalCautionary
AccidentalPlacement
AccidentalSuggestion
Ambitus
AmbitusAccidental
AmbitusLine
AmbitusNoteHead
Arpeggio
BalloonTextItem
BarLine
BarNumber
BassFigure
BassFigureAlignment
BassFigureAlignmentPositioning
BassFigureBracket
BassFigureContinuation
BassFigureLine
Beam
BendAfter
BreakAlignGroup
BreakAlignment
BreathingSign
ChordName
Clef
ClusterSpanner
ClusterSpannerBeacon
CombineTextScript
CueClef
CueEndClef
Custos
DotColumn
Dots
DoublePercentRepeat
DoublePercentRepeatCounter
DoubleRepeatSlash
DynamicLineSpanner
DynamicText
DynamicTextSpanner
Episema
Fingering
FingeringColumn
Flag
FootnoteItem
FootnoteSpanner
FretBoard
Glissando
GraceSpacing
GridLine
GridPoint
Hairpin
HorizontalBracket
InstrumentName
InstrumentSwitch
KeyCancellation
KeySignature
LaissezVibrerTie
LaissezVibrerTieColumn
LedgerLineSpanner
LeftEdge
```

LigatureBracket  
LyricExtender  
LyricHyphen  
LyricSpace  
LyricText  
MeasureCounter  
MeasureGrouping  
MelodyItem  
MensuralLigature  
MetronomeMark  
MultiMeasureRest  
MultiMeasureRestNumber  
MultiMeasureRestText  
NonMusicalPaperColumn  
NoteCollision  
NoteColumn  
NoteHead  
NoteName  
NoteSpacing  
OctavateEight  
OttavaBracket  
PaperColumn  
ParenthesesItem  
PercentRepeat  
PercentRepeatCounter  
PhrasingSlur  
PianoPedalBracket  
RehearsalMark  
RepeatSlash  
RepeatTie  
RepeatTieColumn  
Rest  
RestCollision  
Script  
ScriptColumn  
ScriptRow  
Slur  
SostenutoPedal  
SostenutoPedalLineSpanner  
SpacingSpanner  
SpanBar  
SpanBarStub  
StaffGrouper  
StaffSpacing  
StaffSymbol  
StanzaNumber  
Stem  
StemStub  
StemTremolo  
StringNumber  
StrokeFinger  
SustainPedal  
SustainPedalLineSpanner  
System  
SystemStartBar  
SystemStartBrace  
SystemStartBracket  
SystemStartSquare  
TabNoteHead  
TextScript  
TextSpanner  
Tie  
TieColumn  
TimeSignature  
TrillPitchAccidental  
TrillPitchGroup  
TrillPitchHead  
TrillSpanner  
TupletBracket  
TupletNumber  
UnaCordaPedal  
UnaCordaPedalLineSpanner  
VaticanaLigature



```

VerticalAlignment
VerticalAxisGroup
VoiceFollower
VoltaBracket
VoltaBracketSpanner

```

Returns tuple.

`LilyPondParser.list_known_languages()`

List all note-input languages recognized by LilyPondParser:

```

>>> for x in lilypondparsertools.LilyPondParser.list_known_languages():
...     print x
...
catalan
deutsch
english
espanol
español
français
italiano
nederlands
norsk
portugues
suomi
svenska
vlaams

```

Returns list.

`LilyPondParser.list_known_markup_functions()`

List all markup functions recognized by LilyPondParser:

```

>>> for x in lilypondparsertools.LilyPondParser.list_known_markup_functions():
...     print x
...
abs-fontsize
arrow-head
auto-footnote
backslashed-digit
beam
bold
box
bracket
caps
center-align
center-column
char
circle
column
column-lines
combine
concat
customTabClef
dir-column
doubleflat
doublesharp
draw-circle
draw-hline
draw-line
dynamic
epsfile
eyeglasses
fill-line
fill-with-pattern
filled-box
finger
flat
fontCaps
fontsize
footnote
fraction

```

fret-diagram  
fret-diagram-terse  
fret-diagram-verbose  
fromproperty  
general-align  
halign  
harp-pedal  
hbracket  
hcenter-in  
hspace  
huge  
italic  
justified-lines  
justify  
justify-field  
justify-string  
large  
larger  
left-align  
left-brace  
left-column  
line  
lookup  
lower  
magnify  
markalphabet  
markletter  
medium  
musicglyph  
natural  
normal-size-sub  
normal-size-super  
normal-text  
normalsize  
note  
note-by-number  
null  
number  
on-the-fly  
override  
override-lines  
pad  
pad-around  
pad-to-box  
pad-x  
page-link  
page-ref  
parenthesize  
path  
pattern  
postscript  
property-recursive  
put-adjacent  
raise  
replace  
rest  
rest-by-number  
right-align  
right-brace  
right-column  
roman  
rotate  
rounded-box  
sans  
scale  
score  
semiflat  
semisharp  
sesquiflat  
sesquisharp  
sharp  
simple  
slashed-digit

```

small
smallCaps
smaller
stencil
strut
sub
super
table-of-contents
teeny
text
tied-lyric
tiny
translate
translate-scaled
transparent
triangle
typewriter
underline
upright
vcenter
verbatim-file
vspace
whiteout
with-color
with-dimensions
with-link
with-url
woodwind-diagram
wordwrap
wordwrap-field
wordwrap-internal
wordwrap-lines
wordwrap-string
wordwrap-string-internal

```

Returns list.

`LilyPondParser.list_known_music_functions()`

List all music functions recognized by LilyPondParser:

```

>>> for x in lilypondparsertools.LilyPondParser.list_known_music_functions():
...     print x
...
acciaccatura
appoggiatura
bar
breathe
clef
grace
key
language
makeClusters
mark
relative
skip
time
times
transpose

```

Returns list.

## Special methods

`LilyPondParser.__call__(input_string)`

`(AbjadObject).__eq__(expr)`

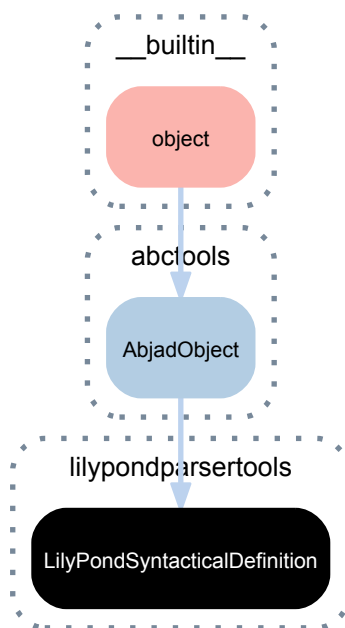
True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

(AbjadObject).**\_\_repr\_\_**()  
Interpreter representation of Abjad object.  
Returns string.

### 59.2.9 lilypondparsertools.LilyPondSyntacticalDefinition



**class** `lilypondparsertools.LilyPondSyntacticalDefinition` (*client*)  
The syntactical definition of LilyPond's syntax.  
Effectively equivalent to LilyPond's `parser.yy` file.  
Not composer-safe.  
Used internally by `LilyPondParser`.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Methods

`LilyPondSyntacticalDefinition.p_assignment__assignment_id__Chr61__identifier_init` (*p*)  
assignment : assignment\_id '=' identifier\_init

`LilyPondSyntacticalDefinition.p_assignment__embedded_scm` (*p*)  
assignment : embedded\_scm

`LilyPondSyntacticalDefinition.p_assignment_id__STRING` (*p*)  
assignment\_id : STRING

`LilyPondSyntacticalDefinition.p_bare_number__REAL__NUMBER_IDENTIFIER` (*p*)  
bare\_number : REAL NUMBER\_IDENTIFIER

LilyPondSyntacticalDefinition.p\_bare\_number\_\_UNSIGNED\_\_NUMBER\_IDENTIFIER (p)  
bare\_number : UNSIGNED NUMBER\_IDENTIFIER

LilyPondSyntacticalDefinition.p\_bare\_number\_\_bare\_number\_closed (p)  
bare\_number : bare\_number\_closed

LilyPondSyntacticalDefinition.p\_bare\_number\_closed\_\_NUMBER\_IDENTIFIER (p)  
bare\_number\_closed : NUMBER\_IDENTIFIER

LilyPondSyntacticalDefinition.p\_bare\_number\_closed\_\_REAL (p)  
bare\_number\_closed : REAL

LilyPondSyntacticalDefinition.p\_bare\_number\_closed\_\_UNSIGNED (p)  
bare\_number\_closed : UNSIGNED

LilyPondSyntacticalDefinition.p\_bare\_unsigned\_\_UNSIGNED (p)  
bare\_unsigned : UNSIGNED

LilyPondSyntacticalDefinition.p\_braced\_music\_list\_\_Chr123\_\_music\_list\_\_Chr125 (p)  
braced\_music\_list : '{ music\_list '}

LilyPondSyntacticalDefinition.p\_chord\_body\_\_ANGLE\_OPEN\_\_chord\_body\_elements\_\_ANGLE\_CLOSE  
chord\_body : ANGLE\_OPEN chord\_body\_elements ANGLE\_CLOSE

LilyPondSyntacticalDefinition.p\_chord\_body\_element\_\_music\_function\_chord\_body (p)  
chord\_body\_element : music\_function\_chord\_body

LilyPondSyntacticalDefinition.p\_chord\_body\_element\_\_pitch\_\_exclamations\_\_questions\_\_octave\_\_check\_\_post\_events  
chord\_body\_element : pitch exclamations questions octave\_check post\_events

LilyPondSyntacticalDefinition.p\_chord\_body\_elements\_\_Empty (p)  
chord\_body\_elements :

LilyPondSyntacticalDefinition.p\_chord\_body\_elements\_\_chord\_body\_elements\_\_chord\_body\_element  
chord\_body\_elements : chord\_body\_elements chord\_body\_element

LilyPondSyntacticalDefinition.p\_closed\_music\_\_complex\_music\_prefix\_\_closed\_music (p)  
closed\_music : complex\_music\_prefix closed\_music

LilyPondSyntacticalDefinition.p\_closed\_music\_\_music\_bare (p)  
closed\_music : music\_bare

LilyPondSyntacticalDefinition.p\_command\_element\_\_Chr124 (p)  
command\_element : 'l'

LilyPondSyntacticalDefinition.p\_command\_element\_\_E\_BACKSLASH (p)  
command\_element : E\_BACKSLASH

LilyPondSyntacticalDefinition.p\_command\_element\_\_command\_event (p)  
command\_element : command\_event

LilyPondSyntacticalDefinition.p\_command\_event\_\_tempo\_event (p)  
command\_event : tempo\_event

LilyPondSyntacticalDefinition.p\_complex\_music\_\_complex\_music\_prefix\_\_music (p)  
complex\_music : complex\_music\_prefix music

LilyPondSyntacticalDefinition.p\_complex\_music\_\_music\_function\_call (p)  
complex\_music : music\_function\_call

LilyPondSyntacticalDefinition.p\_complex\_music\_prefix\_\_CONTEXT\_\_simple\_string\_\_optional\_id\_\_optional\_context\_mod  
complex\_music\_prefix : CONTEXT simple\_string optional\_id optional\_context\_mod

LilyPondSyntacticalDefinition.p\_complex\_music\_prefix\_\_NEWCONTEXT\_\_simple\_string\_\_optional\_id\_\_optional\_context\_mod  
complex\_music\_prefix : NEWCONTEXT simple\_string optional\_id optional\_context\_mod

LilyPondSyntacticalDefinition.p\_composite\_music\_\_complex\_music (p)  
composite\_music : complex\_music

```
LilyPondSyntacticalDefinition.p_direction_less_char__Chr91 (p)
  direction_less_char : ']'
```

---

```

LilyPondSyntacticalDefinition.p_direction_less_char__Chr93 (p)
    direction_less_char : ']'

LilyPondSyntacticalDefinition.p_direction_less_char__E_ANGLE_CLOSE (p)
    direction_less_char : E_ANGLE_CLOSE

LilyPondSyntacticalDefinition.p_direction_less_char__E_ANGLE_OPEN (p)
    direction_less_char : E_ANGLE_OPEN

LilyPondSyntacticalDefinition.p_direction_less_char__E_CLOSE (p)
    direction_less_char : E_CLOSE

LilyPondSyntacticalDefinition.p_direction_less_char__E_EXCLAMATION (p)
    direction_less_char : E_EXCLAMATION

LilyPondSyntacticalDefinition.p_direction_less_char__E_OPEN (p)
    direction_less_char : E_OPEN

LilyPondSyntacticalDefinition.p_direction_less_event__EVENT_IDENTIFIER (p)
    direction_less_event : EVENT_IDENTIFIER

LilyPondSyntacticalDefinition.p_direction_less_event__direction_less_char (p)
    direction_less_event : direction_less_char

LilyPondSyntacticalDefinition.p_direction_less_event__event_function_event (p)
    direction_less_event : event_function_event

LilyPondSyntacticalDefinition.p_direction_less_event__tremolo_type (p)
    direction_less_event : tremolo_type

LilyPondSyntacticalDefinition.p_direction_reqd_event__gen_text_def (p)
    direction_reqd_event : gen_text_def

LilyPondSyntacticalDefinition.p_direction_reqd_event__script_abbreviation (p)
    direction_reqd_event : script_abbreviation

LilyPondSyntacticalDefinition.p_dots__Empty (p)
    dots :

LilyPondSyntacticalDefinition.p_dots__dots__Chr46 (p)
    dots : dots '.'

LilyPondSyntacticalDefinition.p_duration_length__multiplied_duration (p)
    duration_length : multiplied_duration

LilyPondSyntacticalDefinition.p_embedded_scm__embedded_scm_bare (p)
    embedded_scm : embedded_scm_bare

LilyPondSyntacticalDefinition.p_embedded_scm__scm_function_call (p)
    embedded_scm : scm_function_call

LilyPondSyntacticalDefinition.p_embedded_scm_arg__embedded_scm_bare_arg (p)
    embedded_scm_arg : embedded_scm_bare_arg

LilyPondSyntacticalDefinition.p_embedded_scm_arg__music_arg (p)
    embedded_scm_arg : music_arg

LilyPondSyntacticalDefinition.p_embedded_scm_arg__scm_function_call (p)
    embedded_scm_arg : scm_function_call

LilyPondSyntacticalDefinition.p_embedded_scm_arg_closed__closed_music (p)
    embedded_scm_arg_closed : closed_music

LilyPondSyntacticalDefinition.p_embedded_scm_arg_closed__embedded_scm_bare_arg (p)
    embedded_scm_arg_closed : embedded_scm_bare_arg

LilyPondSyntacticalDefinition.p_embedded_scm_arg_closed__scm_function_call_closed (p)
    embedded_scm_arg_closed : scm_function_call_closed

```

```

LilyPondSyntacticalDefinition.p_embedded_scm_bare__SCM_IDENTIFIER (p)
    embedded_scm_bare : SCM_IDENTIFIER

LilyPondSyntacticalDefinition.p_embedded_scm_bare__SCM_TOKEN (p)
    embedded_scm_bare : SCM_TOKEN

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__STRING (p)
    embedded_scm_bare_arg : STRING

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__STRING_IDENTIFIER (p)
    embedded_scm_bare_arg : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__context_def_spec_block (p)
    embedded_scm_bare_arg : context_def_spec_block

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__context_modification (p)
    embedded_scm_bare_arg : context_modification

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__embedded_scm_bare (p)
    embedded_scm_bare_arg : embedded_scm_bare

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__full_markup (p)
    embedded_scm_bare_arg : full_markup

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__full_markup_list (p)
    embedded_scm_bare_arg : full_markup_list

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__output_def (p)
    embedded_scm_bare_arg : output_def

LilyPondSyntacticalDefinition.p_embedded_scm_bare_arg__score_block (p)
    embedded_scm_bare_arg : score_block

LilyPondSyntacticalDefinition.p_embedded_scm_chord_body__SCM_FUNCTION__music_function_chord_body_arglist (p)
    embedded_scm_chord_body : SCM_FUNCTION music_function_chord_body_arglist

LilyPondSyntacticalDefinition.p_embedded_scm_chord_body__bare_number (p)
    embedded_scm_chord_body : bare_number

LilyPondSyntacticalDefinition.p_embedded_scm_chord_body__chord_body_element (p)
    embedded_scm_chord_body : chord_body_element

LilyPondSyntacticalDefinition.p_embedded_scm_chord_body__embedded_scm_bare_arg (p)
    embedded_scm_chord_body : embedded_scm_bare_arg

LilyPondSyntacticalDefinition.p_embedded_scm_chord_body__fraction (p)
    embedded_scm_chord_body : fraction

LilyPondSyntacticalDefinition.p_embedded_scm_closed__embedded_scm_bare (p)
    embedded_scm_closed : embedded_scm_bare

LilyPondSyntacticalDefinition.p_embedded_scm_closed__scm_function_call_closed (p)
    embedded_scm_closed : scm_function_call_closed

LilyPondSyntacticalDefinition.p_error (p)

LilyPondSyntacticalDefinition.p_event_chord__CHORD_REPETITION__optional_notemode_duration__post_events (p)
    event_chord : CHORD_REPETITION optional_notemode_duration post_events

LilyPondSyntacticalDefinition.p_event_chord__MULTI_MEASURE_REST__optional_notemode_duration__post_events (p)
    event_chord : MULTI_MEASURE_REST optional_notemode_duration post_events

LilyPondSyntacticalDefinition.p_event_chord__command_element (p)
    event_chord : command_element

LilyPondSyntacticalDefinition.p_event_chord__note_chord_element (p)
    event_chord : note_chord_element

LilyPondSyntacticalDefinition.p_event_chord__simple_chord_elements__post_events (p)
    event_chord : simple_chord_elements post_events

```



---

```

LilyPondSyntacticalDefinition.p_event_function_event__EVENT_FUNCTION__function_arglist_closed
    event_function_event : EVENT_FUNCTION function_arglist_closed

LilyPondSyntacticalDefinition.p_exclamations__Empty (p)
    exclamations :

LilyPondSyntacticalDefinition.p_exclamations__exclamations__Chr33 (p)
    exclamations : exclamations '!'

LilyPondSyntacticalDefinition.p_fingering__UNSIGNED (p)
    fingering : UNSIGNED

LilyPondSyntacticalDefinition.p_fraction__FRACTION (p)
    fraction : FRACTION

LilyPondSyntacticalDefinition.p_fraction__UNSIGNED__Chr47__UNSIGNED (p)
    fraction : UNSIGNED '/' UNSIGNED

LilyPondSyntacticalDefinition.p_full_markup__MARKUP_IDENTIFIER (p)
    full_markup : MARKUP_IDENTIFIER

LilyPondSyntacticalDefinition.p_full_markup__MARKUP__markup_top (p)
    full_markup : MARKUP markup_top

LilyPondSyntacticalDefinition.p_full_markup_list__MARKUPLIST_IDENTIFIER (p)
    full_markup_list : MARKUPLIST_IDENTIFIER

LilyPondSyntacticalDefinition.p_full_markup_list__MARKUPLIST__markup_list (p)
    full_markup_list : MARKUPLIST markup_list

LilyPondSyntacticalDefinition.p_function_arglist__function_arglist_common (p)
    function_arglist : function_arglist_common

LilyPondSyntacticalDefinition.p_function_arglist__function_arglist_nonbackup (p)
    function_arglist : function_arglist_nonbackup

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_DURATION__function_arglist_closed_keep_duration_length
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_closed_keep duration_length

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_PITCH__function_arglist_closed_keep_pitch_also_in_chords
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_closed_keep pitch_also_in_chords

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_keep_backup
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep BACKUP

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_keep_minus_NUM-BER_IDENTIFIER
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' NUM-BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_keep_minus_REAL
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_keep_minus_UNSIGNED
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' UNSIGNED

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_keep_FRACTION
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep FRACTION

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_keep_NUM-BER_IDENTIFIER
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep NUM-BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_keep_REAL
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep REAL

```

```

LilyPondSyntacticalDefinition.p_function_arglist_backup_EXPECT_OPTIONAL_EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep UN-
        SIGNED

LilyPondSyntacticalDefinition.p_function_arglist_backup_EXPECT_OPTIONAL_EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep
        post_event_nofinger

LilyPondSyntacticalDefinition.p_function_arglist_backup_EXPECT_OPTIONAL_EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_keep embed-
        ded_scm_arg_closed

LilyPondSyntacticalDefinition.p_function_arglist_backup_function_arglist_backup_REPARS
    function_arglist_backup : function_arglist_backup REPARSE bare_number

LilyPondSyntacticalDefinition.p_function_arglist_backup_function_arglist_backup_REPARS
    function_arglist_backup : function_arglist_backup REPARSE embedded_scm_arg_closed

LilyPondSyntacticalDefinition.p_function_arglist_backup_function_arglist_backup_REPARS
    function_arglist_backup : function_arglist_backup REPARSE fraction

LilyPondSyntacticalDefinition.p_function_arglist_bare_EXPECT_DURATION_function_arglist
    function_arglist_bare : EXPECT_DURATION function_arglist_closed_optional duration_length

LilyPondSyntacticalDefinition.p_function_arglist_bare_EXPECT_NO_MORE_ARGS(p)
    function_arglist_bare : EXPECT_NO_MORE_ARGS

LilyPondSyntacticalDefinition.p_function_arglist_bare_EXPECT_OPTIONAL_EXPECT_DURATION_
    function_arglist_bare : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_skip DEFAULT

LilyPondSyntacticalDefinition.p_function_arglist_bare_EXPECT_OPTIONAL_EXPECT_PITCH__fu
    function_arglist_bare : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_skip DEFAULT

LilyPondSyntacticalDefinition.p_function_arglist_bare_EXPECT_OPTIONAL_EXPECT_SCM__func
    function_arglist_bare : EXPECT_OPTIONAL EXPECT_SCM function_arglist_skip DEFAULT

LilyPondSyntacticalDefinition.p_function_arglist_bare_EXPECT_PITCH_function_arglist_op
    function_arglist_bare : EXPECT_PITCH function_arglist_optional pitch_also_in_chords

LilyPondSyntacticalDefinition.p_function_arglist_closed_function_arglist_closed_common
    function_arglist_closed : function_arglist_closed_common

LilyPondSyntacticalDefinition.p_function_arglist_closed_function_arglist_nonbackup(p)
    function_arglist_closed : function_arglist_nonbackup

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
    function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' NUM-
        BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
    function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
    function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' UNSIGNED

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
    function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional bare_number

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
    function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional fraction

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
    function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional post_event_nofinger

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
    function_arglist_closed_common : EXPECT_SCM function_arglist_optional embedded_scm_arg_closed

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_function_arglist_bare(p)
    function_arglist_closed_common : function_arglist_bare

```

LilyPondSyntacticalDefinition.p\_function\_arglist\_closed\_keep\_\_function\_arglist\_backup (p)  
function\_arglist\_closed\_keep : function\_arglist\_backup

LilyPondSyntacticalDefinition.p\_function\_arglist\_closed\_keep\_\_function\_arglist\_closed\_common  
function\_arglist\_closed\_keep : function\_arglist\_closed\_common

LilyPondSyntacticalDefinition.p\_function\_arglist\_closed\_optional\_\_EXPECT\_OPTIONAL\_\_EXPECT\_DURATION\_\_EXPECT\_DURATION\_\_function\_arglist\_closed\_optional : EXPECT\_OPTIONAL EXPECT\_DURATION func-

LilyPondSyntacticalDefinition.p\_function\_arglist\_closed\_optional\_\_EXPECT\_OPTIONAL\_\_EXPECT\_DURATION\_\_EXPECT\_DURATION\_\_function\_arglist\_closed\_optional : EXPECT\_OPTIONAL EXPECT\_PITCH func-

LilyPondSyntacticalDefinition.p\_function\_arglist\_closed\_optional\_\_function\_arglist\_backup\_\_function\_arglist\_closed\_optional : function\_arglist\_backup BACKUP

LilyPondSyntacticalDefinition.p\_function\_arglist\_closed\_optional\_\_function\_arglist\_closed\_keep\_\_function\_arglist\_closed\_keep : function\_arglist\_closed\_keep %prec FUNCTION\_ARGLIST

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_\_EXPECT\_SCM\_\_function\_arglist\_closed\_optional\_\_function\_arglist\_closed\_optional : EXPECT\_SCM function\_arglist\_closed\_optional bare\_number

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_\_EXPECT\_SCM\_\_function\_arglist\_closed\_optional\_\_function\_arglist\_closed\_optional : EXPECT\_SCM function\_arglist\_closed\_optional fraction

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_\_EXPECT\_SCM\_\_function\_arglist\_closed\_optional\_\_function\_arglist\_closed\_optional : EXPECT\_SCM function\_arglist\_closed\_optional post\_event\_nofinger

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_\_EXPECT\_SCM\_\_function\_arglist\_closed\_optional\_\_function\_arglist\_closed\_optional : EXPECT\_SCM function\_arglist\_optional embedded\_scm\_arg

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_\_function\_arglist\_bare (p)  
function\_arglist\_common : function\_arglist\_bare

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_\_function\_arglist\_common\_minus (p)  
function\_arglist\_common : function\_arglist\_common\_minus

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_minus\_\_EXPECT\_SCM\_\_function\_arglist\_closed\_optional\_\_function\_arglist\_closed\_optional : EXPECT\_SCM function\_arglist\_closed\_optional '-' NUM-BER\_IDENTIFIER

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_minus\_\_EXPECT\_SCM\_\_function\_arglist\_closed\_optional\_\_function\_arglist\_closed\_optional : EXPECT\_SCM function\_arglist\_closed\_optional '-' REAL

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_minus\_\_EXPECT\_SCM\_\_function\_arglist\_closed\_optional\_\_function\_arglist\_closed\_optional : EXPECT\_SCM function\_arglist\_closed\_optional '-' UNSIGNED

LilyPondSyntacticalDefinition.p\_function\_arglist\_common\_minus\_\_function\_arglist\_common\_minus\_\_function\_arglist\_common\_minus : function\_arglist\_common\_minus REPARSE bare\_number

LilyPondSyntacticalDefinition.p\_function\_arglist\_keep\_\_function\_arglist\_backup (p)  
function\_arglist\_keep : function\_arglist\_backup

LilyPondSyntacticalDefinition.p\_function\_arglist\_keep\_\_function\_arglist\_common (p)  
function\_arglist\_keep : function\_arglist\_common

LilyPondSyntacticalDefinition.p\_function\_arglist\_nonbackup\_\_EXPECT\_OPTIONAL\_\_EXPECT\_DURATION\_\_EXPECT\_DURATION\_\_function\_arglist\_nonbackup : EXPECT\_OPTIONAL EXPECT\_DURATION function\_arglist\_closed duration\_length

LilyPondSyntacticalDefinition.p\_function\_arglist\_nonbackup\_\_EXPECT\_OPTIONAL\_\_EXPECT\_DURATION\_\_EXPECT\_DURATION\_\_function\_arglist\_nonbackup : EXPECT\_OPTIONAL EXPECT\_PITCH function\_arglist pitch\_also\_in\_chords

LilyPondSyntacticalDefinition.p\_function\_arglist\_nonbackup\_\_EXPECT\_OPTIONAL\_\_EXPECT\_SCM\_\_function\_arglist\_nonbackup : EXPECT\_OPTIONAL EXPECT\_SCM function\_arglist embedded\_scm\_arg\_closed

```

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' NUM-BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' UNSIGNED

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed FRACTION

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed bare_number_closed

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed post_event_nofinger

LilyPondSyntacticalDefinition.p_function_arglist_optional__EXPECT_OPTIONAL__EXPECT_DURATION__function_arglist_optional : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_optional

LilyPondSyntacticalDefinition.p_function_arglist_optional__EXPECT_OPTIONAL__EXPECT_PITCH__function_arglist_optional : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_optional

LilyPondSyntacticalDefinition.p_function_arglist_optional__function_arglist_backup__BACKUP__function_arglist_optional : function_arglist_backup BACKUP

LilyPondSyntacticalDefinition.p_function_arglist_optional__function_arglist_keep(p) function_arglist_optional : function_arglist_keep %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_function_arglist_skip__EXPECT_OPTIONAL__EXPECT_DURATION__function_arglist_skip : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_function_arglist_skip__EXPECT_OPTIONAL__EXPECT_PITCH__function_arglist_skip : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_function_arglist_skip__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_skip : EXPECT_OPTIONAL EXPECT_SCM function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_function_arglist_skip__function_arglist_common(p) function_arglist_skip : function_arglist_common

LilyPondSyntacticalDefinition.p_gen_text_def__full_markup(p) gen_text_def : full_markup

LilyPondSyntacticalDefinition.p_gen_text_def__simple_string(p) gen_text_def : simple_string

LilyPondSyntacticalDefinition.p_grouped_music_list__sequential_music(p) grouped_music_list : sequential_music

LilyPondSyntacticalDefinition.p_grouped_music_list__simultaneous_music(p) grouped_music_list : simultaneous_music

LilyPondSyntacticalDefinition.p_identifier_init__context_def_spec_block(p) identifier_init : context_def_spec_block

LilyPondSyntacticalDefinition.p_identifier_init__context_modification(p) identifier_init : context_modification

```

---

```

LilyPondSyntacticalDefinition.p_identifier_init__embedded_scm (p)
    identifier_init : embedded_scm

LilyPondSyntacticalDefinition.p_identifier_init__full_markup (p)
    identifier_init : full_markup

LilyPondSyntacticalDefinition.p_identifier_init__full_markup_list (p)
    identifier_init : full_markup_list

LilyPondSyntacticalDefinition.p_identifier_init__music (p)
    identifier_init : music

LilyPondSyntacticalDefinition.p_identifier_init__number_expression (p)
    identifier_init : number_expression

LilyPondSyntacticalDefinition.p_identifier_init__output_def (p)
    identifier_init : output_def

LilyPondSyntacticalDefinition.p_identifier_init__post_event_nofinger (p)
    identifier_init : post_event_nofinger

LilyPondSyntacticalDefinition.p_identifier_init__score_block (p)
    identifier_init : score_block

LilyPondSyntacticalDefinition.p_identifier_init__string (p)
    identifier_init : string

LilyPondSyntacticalDefinition.p_lilypond__Empty (p)
    lilypond :

LilyPondSyntacticalDefinition.p_lilypond__lilypond__assignment (p)
    lilypond : lilypond assignment

LilyPondSyntacticalDefinition.p_lilypond__lilypond__error (p)
    lilypond : lilypond error

LilyPondSyntacticalDefinition.p_lilypond__lilypond__toplevel_expression (p)
    lilypond : lilypond toplevel_expression

LilyPondSyntacticalDefinition.p_lilypond_header__HEADER__Chr123__lilypond_header_body__Chr123 (p)
    lilypond_header : HEADER {' lilypond_header_body '}

LilyPondSyntacticalDefinition.p_lilypond_header_body__Empty (p)
    lilypond_header_body :

LilyPondSyntacticalDefinition.p_lilypond_header_body__lilypond_header_body__assignment (p)
    lilypond_header_body : lilypond_header_body assignment

LilyPondSyntacticalDefinition.p_markup__markup_head_1_list__simple_markup (p)
    markup : markup_head_1_list simple_markup

LilyPondSyntacticalDefinition.p_markup__simple_markup (p)
    markup : simple_markup

LilyPondSyntacticalDefinition.p_markup_braced_list__Chr123__markup_braced_list_body__Chr123 (p)
    markup_braced_list : {' markup_braced_list_body '}

LilyPondSyntacticalDefinition.p_markup_braced_list_body__Empty (p)
    markup_braced_list_body :

LilyPondSyntacticalDefinition.p_markup_braced_list_body__markup_braced_list_body__markup (p)
    markup_braced_list_body : markup_braced_list_body markup

LilyPondSyntacticalDefinition.p_markup_braced_list_body__markup_braced_list_body__markup_list (p)
    markup_braced_list_body : markup_braced_list_body markup_list

LilyPondSyntacticalDefinition.p_markup_command_basic_arguments__EXPECT_MARKUP_LIST__markup_command_list_arguments (p)
    markup_command_basic_arguments : EXPECT_MARKUP_LIST markup_command_list_arguments
    markup_list

```

```

LilyPondSyntacticalDefinition.p_markup_command_basic_arguments__EXPECT_NO_MORE_ARGS (p)
    markup_command_basic_arguments : EXPECT_NO_MORE_ARGS

LilyPondSyntacticalDefinition.p_markup_command_basic_arguments__EXPECT_SCM_markup_comma
    markup_command_basic_arguments : EXPECT_SCM markup_command_list_arguments embed-
    ded_scm_closed

LilyPondSyntacticalDefinition.p_markup_command_list__MARKUP_LIST_FUNCTION_markup_comma
    markup_command_list : MARKUP_LIST_FUNCTION markup_command_list_arguments

LilyPondSyntacticalDefinition.p_markup_command_list_arguments__EXPECT_MARKUP_markup_com
    markup_command_list_arguments : EXPECT_MARKUP markup_command_list_arguments markup

LilyPondSyntacticalDefinition.p_markup_command_list_arguments__markup_command_basic_argu
    markup_command_list_arguments : markup_command_basic_arguments

LilyPondSyntacticalDefinition.p_markup_composed_list__markup_head_1_list__markup_braced_
    markup_composed_list : markup_head_1_list markup_braced_list

LilyPondSyntacticalDefinition.p_markup_head_1_item__MARKUP_FUNCTION__EXPECT_MARKUP__mark
    markup_head_1_item : MARKUP_FUNCTION EXPECT_MARKUP markup_command_list_arguments

LilyPondSyntacticalDefinition.p_markup_head_1_list__markup_head_1_item (p)
    markup_head_1_list : markup_head_1_item

LilyPondSyntacticalDefinition.p_markup_head_1_list__markup_head_1_list__markup_head_1_it
    markup_head_1_list : markup_head_1_list markup_head_1_item

LilyPondSyntacticalDefinition.p_markup_list__MARKUPLIST_IDENTIFIER (p)
    markup_list : MARKUPLIST_IDENTIFIER

LilyPondSyntacticalDefinition.p_markup_list__markup_braced_list (p)
    markup_list : markup_braced_list

LilyPondSyntacticalDefinition.p_markup_list__markup_command_list (p)
    markup_list : markup_command_list

LilyPondSyntacticalDefinition.p_markup_list__markup_composed_list (p)
    markup_list : markup_composed_list

LilyPondSyntacticalDefinition.p_markup_list__markup_scm__MARKUPLIST_IDENTIFIER (p)
    markup_list : markup_scm MARKUPLIST_IDENTIFIER

LilyPondSyntacticalDefinition.p_markup_scm__embedded_scm_bare__BACKUP (p)
    markup_scm : embedded_scm_bare BACKUP

LilyPondSyntacticalDefinition.p_markup_top__markup_head_1_list__simple_markup (p)
    markup_top : markup_head_1_list simple_markup

LilyPondSyntacticalDefinition.p_markup_top__markup_list (p)
    markup_top : markup_list

LilyPondSyntacticalDefinition.p_markup_top__simple_markup (p)
    markup_top : simple_markup

LilyPondSyntacticalDefinition.p_multiplied_duration__multiplied_duration__Chr42__FRACTIO
    multiplied_duration : multiplied_duration "*" FRACTION

LilyPondSyntacticalDefinition.p_multiplied_duration__multiplied_duration__Chr42__bare_un
    multiplied_duration : multiplied_duration "*" bare_unsigned

LilyPondSyntacticalDefinition.p_multiplied_duration__steno_duration (p)
    multiplied_duration : steno_duration

LilyPondSyntacticalDefinition.p_music__composite_music (p)
    music : composite_music %prec COMPOSITE

LilyPondSyntacticalDefinition.p_music__simple_music (p)
    music : simple_music

```

---

```

LilyPondSyntacticalDefinition.p_music_arg__composite_music (p)
    music_arg : composite_music %prec COMPOSITE

LilyPondSyntacticalDefinition.p_music_arg__simple_music (p)
    music_arg : simple_music

LilyPondSyntacticalDefinition.p_music_bare__MUSIC_IDENTIFIER (p)
    music_bare : MUSIC_IDENTIFIER

LilyPondSyntacticalDefinition.p_music_bare__grouped_music_list (p)
    music_bare : grouped_music_list

LilyPondSyntacticalDefinition.p_music_function_call__MUSIC_FUNCTION__function_arglist (p)
    music_function_call : MUSIC_FUNCTION function_arglist

LilyPondSyntacticalDefinition.p_music_function_chord_body__MUSIC_FUNCTION__music_function_chord_body_arglist (p)
    music_function_chord_body : MUSIC_FUNCTION music_function_chord_body_arglist

LilyPondSyntacticalDefinition.p_music_function_chord_body_arglist__EXPECT_SCM__music_function_chord_body_arglist (p)
    music_function_chord_body_arglist : EXPECT_SCM music_function_chord_body_arglist embed-
    ded_scm_chord_body

LilyPondSyntacticalDefinition.p_music_function_chord_body_arglist__function_arglist_bare (p)
    music_function_chord_body_arglist : function_arglist_bare

LilyPondSyntacticalDefinition.p_music_function_event__MUSIC_FUNCTION__function_arglist_closed (p)
    music_function_event : MUSIC_FUNCTION function_arglist_closed

LilyPondSyntacticalDefinition.p_music_list__Empty (p)
    music_list :

LilyPondSyntacticalDefinition.p_music_list__music_list__embedded_scm (p)
    music_list : music_list embedded_scm

LilyPondSyntacticalDefinition.p_music_list__music_list__error (p)
    music_list : music_list error

LilyPondSyntacticalDefinition.p_music_list__music_list__music (p)
    music_list : music_list music

LilyPondSyntacticalDefinition.p_music_property_def__simple_music_property_def (p)
    music_property_def : simple_music_property_def

LilyPondSyntacticalDefinition.p_note_chord_element__chord_body__optional_notemode_duration__post_events (p)
    note_chord_element : chord_body optional_notemode_duration post_events

LilyPondSyntacticalDefinition.p_number_expression__number_expression__Chr43__number_term (p)
    number_expression : number_expression '+' number_term

LilyPondSyntacticalDefinition.p_number_expression__number_expression__Chr45__number_term (p)
    number_expression : number_expression '-' number_term

LilyPondSyntacticalDefinition.p_number_expression__number_term (p)
    number_expression : number_term

LilyPondSyntacticalDefinition.p_number_factor__Chr45__number_factor (p)
    number_factor : '-' number_factor

LilyPondSyntacticalDefinition.p_number_factor__bare_number (p)
    number_factor : bare_number

LilyPondSyntacticalDefinition.p_number_term__number_factor (p)
    number_term : number_factor

LilyPondSyntacticalDefinition.p_number_term__number_factor__Chr42__number_factor (p)
    number_term : number_factor '*' number_factor

LilyPondSyntacticalDefinition.p_number_term__number_factor__Chr47__number_factor (p)
    number_term : number_factor '/' number_factor

```

```

LilyPondSyntacticalDefinition.p_octave_check__Chr61 (p)
    octave_check : '='

LilyPondSyntacticalDefinition.p_octave_check__Chr61__sub_quotes (p)
    octave_check : '=' sub_quotes

LilyPondSyntacticalDefinition.p_octave_check__Chr61__sup_quotes (p)
    octave_check : '=' sup_quotes

LilyPondSyntacticalDefinition.p_octave_check__Empty (p)
    octave_check :

LilyPondSyntacticalDefinition.p_optional_context_mod__Empty (p)
    optional_context_mod :

LilyPondSyntacticalDefinition.p_optional_context_mod__context_modification (p)
    optional_context_mod : context_modification

LilyPondSyntacticalDefinition.p_optional_id__Chr61__simple_string (p)
    optional_id : '=' simple_string

LilyPondSyntacticalDefinition.p_optional_id__Empty (p)
    optional_id :

LilyPondSyntacticalDefinition.p_optional_notemode_duration__Empty (p)
    optional_notemode_duration :

LilyPondSyntacticalDefinition.p_optional_notemode_duration__multiplied_duration (p)
    optional_notemode_duration : multiplied_duration

LilyPondSyntacticalDefinition.p_optional_rest__Empty (p)
    optional_rest :

LilyPondSyntacticalDefinition.p_optional_rest__REST (p)
    optional_rest : REST

LilyPondSyntacticalDefinition.p_output_def__output_def_body__Chr125 (p)
    output_def : output_def_body '}'

LilyPondSyntacticalDefinition.p_output_def_body__output_def_body__assignment (p)
    output_def_body : output_def_body assignment

LilyPondSyntacticalDefinition.p_output_def_body__output_def_head_with_mode_switch__Chr125 (p)
    output_def_body : output_def_head_with_mode_switch '{'

LilyPondSyntacticalDefinition.p_output_def_body__output_def_head_with_mode_switch__Chr125 (p)
    output_def_body : output_def_head_with_mode_switch '{' output_DEF_IDENTIFIER

LilyPondSyntacticalDefinition.p_output_def_head__LAYOUT (p)
    output_def_head : LAYOUT

LilyPondSyntacticalDefinition.p_output_def_head__MIDI (p)
    output_def_head : MIDI

LilyPondSyntacticalDefinition.p_output_def_head__PAPER (p)
    output_def_head : PAPER

LilyPondSyntacticalDefinition.p_output_def_head_with_mode_switch__output_def_head (p)
    output_def_head_with_mode_switch : output_def_head

LilyPondSyntacticalDefinition.p_pitch__PITCH_IDENTIFIER (p)
    pitch : PITCH_IDENTIFIER

LilyPondSyntacticalDefinition.p_pitch__steno_pitch (p)
    pitch : steno_pitch

LilyPondSyntacticalDefinition.p_pitch_also_in_chords__pitch (p)
    pitch_also_in_chords : pitch

```



---

```

LilyPondSyntacticalDefinition.p_pitch_also_in_chords__steno_tonic_pitch (p)
    pitch_also_in_chords : steno_tonic_pitch

LilyPondSyntacticalDefinition.p_post_event__Chr45__fingering (p)
    post_event : '-' fingering

LilyPondSyntacticalDefinition.p_post_event__post_event_nofinger (p)
    post_event : post_event_nofinger

LilyPondSyntacticalDefinition.p_post_event_nofinger__Chr94__fingering (p)
    post_event_nofinger : '^' fingering

LilyPondSyntacticalDefinition.p_post_event_nofinger__Chr95__fingering (p)
    post_event_nofinger : '_' fingering

LilyPondSyntacticalDefinition.p_post_event_nofinger__EXTENDER (p)
    post_event_nofinger : EXTENDER

LilyPondSyntacticalDefinition.p_post_event_nofinger__HYPHEN (p)
    post_event_nofinger : HYPHEN

LilyPondSyntacticalDefinition.p_post_event_nofinger__direction_less_event (p)
    post_event_nofinger : direction_less_event

LilyPondSyntacticalDefinition.p_post_event_nofinger__script_dir__direction_less_event (p)
    post_event_nofinger : script_dir direction_less_event

LilyPondSyntacticalDefinition.p_post_event_nofinger__script_dir__direction_reqd_event (p)
    post_event_nofinger : script_dir direction_reqd_event

LilyPondSyntacticalDefinition.p_post_event_nofinger__script_dir__music_function_event (p)
    post_event_nofinger : script_dir music_function_event

LilyPondSyntacticalDefinition.p_post_event_nofinger__string_number_event (p)
    post_event_nofinger : string_number_event

LilyPondSyntacticalDefinition.p_post_events__Empty (p)
    post_events :

LilyPondSyntacticalDefinition.p_post_events__post_events__post_event (p)
    post_events : post_events post_event

LilyPondSyntacticalDefinition.p_property_operation__OVERRIDE__simple_string__property_path__property_operation (p)
    property_operation : OVERRIDE simple_string property_path '=' scalar

LilyPondSyntacticalDefinition.p_property_operation__REVERT__simple_string__embedded_scm__property_operation (p)
    property_operation : REVERT simple_string embedded_scm

LilyPondSyntacticalDefinition.p_property_operation__STRING__Chr61__scalar (p)
    property_operation : STRING '=' scalar

LilyPondSyntacticalDefinition.p_property_operation__UNSET__simple_string (p)
    property_operation : UNSET simple_string

LilyPondSyntacticalDefinition.p_property_path__property_path_revved (p)
    property_path : property_path_revved

LilyPondSyntacticalDefinition.p_property_path_revved__embedded_scm__closed (p)
    property_path_revved : embedded_scm_closed

LilyPondSyntacticalDefinition.p_property_path_revved__property_path_revved__embedded_scm__closed (p)
    property_path_revved : property_path_revved embedded_scm_closed

LilyPondSyntacticalDefinition.p_questions__Empty (p)
    questions :

LilyPondSyntacticalDefinition.p_questions__questions__Chr63 (p)
    questions : questions '?'

```

---

```

LilyPondSyntacticalDefinition.p_scalar__bare_number (p)
    scalar : bare_number

LilyPondSyntacticalDefinition.p_scalar__embedded_scm_arg (p)
    scalar : embedded_scm_arg

LilyPondSyntacticalDefinition.p_scalar_closed__bare_number (p)
    scalar_closed : bare_number

LilyPondSyntacticalDefinition.p_scalar_closed__embedded_scm_arg_closed (p)
    scalar_closed : embedded_scm_arg_closed

LilyPondSyntacticalDefinition.p_scm_function_call__SCM_FUNCTION__function_arglist (p)
    scm_function_call : SCM_FUNCTION function_arglist

LilyPondSyntacticalDefinition.p_scm_function_call_closed__SCM_FUNCTION__function_arglist
    scm_function_call_closed : SCM_FUNCTION function_arglist_closed %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_score_block__SCORE__Chr123__score_body__Chr125 (p)
    score_block : SCORE '{ score_body '}'

LilyPondSyntacticalDefinition.p_score_body__SCORE_IDENTIFIER (p)
    score_body : SCORE_IDENTIFIER

LilyPondSyntacticalDefinition.p_score_body__music (p)
    score_body : music

LilyPondSyntacticalDefinition.p_score_body__score_body__lilypond_header (p)
    score_body : score_body lilypond_header

LilyPondSyntacticalDefinition.p_score_body__score_body__output_def (p)
    score_body : score_body output_def

LilyPondSyntacticalDefinition.p_script_abbreviation__ANGLE_CLOSE (p)
    script_abbreviation : ANGLE_CLOSE

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr124 (p)
    script_abbreviation : '|'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr43 (p)
    script_abbreviation : '+'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr45 (p)
    script_abbreviation : '-'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr46 (p)
    script_abbreviation : '.'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr94 (p)
    script_abbreviation : '^'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr95 (p)
    script_abbreviation : '_'

LilyPondSyntacticalDefinition.p_script_dir__Chr45 (p)
    script_dir : '- '

LilyPondSyntacticalDefinition.p_script_dir__Chr94 (p)
    script_dir : '^'

LilyPondSyntacticalDefinition.p_script_dir__Chr95 (p)
    script_dir : '_'

LilyPondSyntacticalDefinition.p_sequential_music__SEQUENTIAL__braced_music_list (p)
    sequential_music : SEQUENTIAL braced_music_list

LilyPondSyntacticalDefinition.p_sequential_music__braced_music_list (p)
    sequential_music : braced_music_list

```

---

```

LilyPondSyntacticalDefinition.p_simple_chord_elements__simple_element (p)
    simple_chord_elements : simple_element

LilyPondSyntacticalDefinition.p_simple_element__RESTNAME__optional_notemode_duration (p)
    simple_element : RESTNAME optional_notemode_duration

LilyPondSyntacticalDefinition.p_simple_element__pitch__exclamations__questions__octave__check__optional_notemode_duration__optional_rest (p)
    simple_element : pitch exclamations questions octave_check optional_notemode_duration optional_rest

LilyPondSyntacticalDefinition.p_simple_markup__MARKUP_FUNCTION__markup_command_basic_arguments (p)
    simple_markup : MARKUP_FUNCTION markup_command_basic_arguments

LilyPondSyntacticalDefinition.p_simple_markup__MARKUP_IDENTIFIER (p)
    simple_markup : MARKUP_IDENTIFIER

LilyPondSyntacticalDefinition.p_simple_markup__SCORE__Chr123__score_body__Chr125 (p)
    simple_markup : SCORE '{ 'score_body '}'

LilyPondSyntacticalDefinition.p_simple_markup__STRING (p)
    simple_markup : STRING

LilyPondSyntacticalDefinition.p_simple_markup__STRING_IDENTIFIER (p)
    simple_markup : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.p_simple_markup__markup_scm__MARKUP_IDENTIFIER (p)
    simple_markup : markup_scm MARKUP_IDENTIFIER

LilyPondSyntacticalDefinition.p_simple_music__context_change (p)
    simple_music : context_change

LilyPondSyntacticalDefinition.p_simple_music__event_chord (p)
    simple_music : event_chord

LilyPondSyntacticalDefinition.p_simple_music__music_property_def (p)
    simple_music : music_property_def

LilyPondSyntacticalDefinition.p_simple_music_property_def__OVERRIDE__context_prop_spec__property_path__scalar (p)
    simple_music_property_def : OVERRIDE context_prop_spec property_path '=' scalar

LilyPondSyntacticalDefinition.p_simple_music_property_def__REVERT__context_prop_spec__embedded_scm (p)
    simple_music_property_def : REVERT context_prop_spec embedded_scm

LilyPondSyntacticalDefinition.p_simple_music_property_def__SET__context_prop_spec__Chr61__scalar (p)
    simple_music_property_def : SET context_prop_spec '=' scalar

LilyPondSyntacticalDefinition.p_simple_music_property_def__UNSET__context_prop_spec (p)
    simple_music_property_def : UNSET context_prop_spec

LilyPondSyntacticalDefinition.p_simple_string__STRING (p)
    simple_string : STRING

LilyPondSyntacticalDefinition.p_simple_string__STRING_IDENTIFIER (p)
    simple_string : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.p_simultaneous_music__DOUBLE_ANGLE_OPEN__music_list__DOUBLE_ANGLE_CLOSE (p)
    simultaneous_music : DOUBLE_ANGLE_OPEN music_list DOUBLE_ANGLE_CLOSE

LilyPondSyntacticalDefinition.p_simultaneous_music__SIMULTANEOUS__braced_music_list (p)
    simultaneous_music : SIMULTANEOUS braced_music_list

LilyPondSyntacticalDefinition.p_start_symbol__lilypond (p)
    start_symbol : lilypond

LilyPondSyntacticalDefinition.p_steno_duration__DURATION_IDENTIFIER__dots (p)
    steno_duration : DURATION_IDENTIFIER dots

LilyPondSyntacticalDefinition.p_steno_duration__bare_unsigned__dots (p)
    steno_duration : bare_unsigned dots

```

```

LilyPondSyntacticalDefinition.p_steno_pitch__NOTENAME_PITCH (p)
    steno_pitch : NOTENAME_PITCH

LilyPondSyntacticalDefinition.p_steno_pitch__NOTENAME_PITCH__sub_quotes (p)
    steno_pitch : NOTENAME_PITCH sub_quotes

LilyPondSyntacticalDefinition.p_steno_pitch__NOTENAME_PITCH__sup_quotes (p)
    steno_pitch : NOTENAME_PITCH sup_quotes

LilyPondSyntacticalDefinition.p_steno_tonic_pitch__TONICNAME_PITCH (p)
    steno_tonic_pitch : TONICNAME_PITCH

LilyPondSyntacticalDefinition.p_steno_tonic_pitch__TONICNAME_PITCH__sub_quotes (p)
    steno_tonic_pitch : TONICNAME_PITCH sub_quotes

LilyPondSyntacticalDefinition.p_steno_tonic_pitch__TONICNAME_PITCH__sup_quotes (p)
    steno_tonic_pitch : TONICNAME_PITCH sup_quotes

LilyPondSyntacticalDefinition.p_string__STRING (p)
    string : STRING

LilyPondSyntacticalDefinition.p_string__STRING_IDENTIFIER (p)
    string : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.p_string__string__Chr43__string (p)
    string : string '+' string

LilyPondSyntacticalDefinition.p_string_number_event__E_UNSIGNED (p)
    string_number_event : E_UNSIGNED

LilyPondSyntacticalDefinition.p_sub_quotes__Chr44 (p)
    sub_quotes : ','

LilyPondSyntacticalDefinition.p_sub_quotes__sub_quotes__Chr44 (p)
    sub_quotes : sub_quotes ','

LilyPondSyntacticalDefinition.p_sup_quotes__Chr39 (p)
    sup_quotes : '"'

LilyPondSyntacticalDefinition.p_sup_quotes__sup_quotes__Chr39 (p)
    sup_quotes : sup_quotes '"'

LilyPondSyntacticalDefinition.p_tempo_event__TEMPO__scalar (p)
    tempo_event : TEMPO scalar

LilyPondSyntacticalDefinition.p_tempo_event__TEMPO__scalar_closed__steno_duration__Chr61__tempo_range (p)
    tempo_event : TEMPO scalar_closed steno_duration '=' tempo_range

LilyPondSyntacticalDefinition.p_tempo_event__TEMPO__steno_duration__Chr61__tempo_range (p)
    tempo_event : TEMPO steno_duration '=' tempo_range

LilyPondSyntacticalDefinition.p_tempo_range__bare_unsigned (p)
    tempo_range : bare_unsigned

LilyPondSyntacticalDefinition.p_tempo_range__bare_unsigned__Chr45__bare_unsigned (p)
    tempo_range : bare_unsigned '-' bare_unsigned

LilyPondSyntacticalDefinition.p_toplevel_expression__composite_music (p)
    toplevel_expression : composite_music

LilyPondSyntacticalDefinition.p_toplevel_expression__full_markup (p)
    toplevel_expression : full_markup

LilyPondSyntacticalDefinition.p_toplevel_expression__full_markup_list (p)
    toplevel_expression : full_markup_list

LilyPondSyntacticalDefinition.p_toplevel_expression__lilypond_header (p)
    toplevel_expression : lilypond_header

```

```

LilyPondSyntacticalDefinition.p_toplevel_expression__output_def(p)
    toplevel_expression : output_def

LilyPondSyntacticalDefinition.p_toplevel_expression__score_block(p)
    toplevel_expression : score_block

LilyPondSyntacticalDefinition.p_tremolo_type__Chr58(p)
    tremolo_type : ':'

LilyPondSyntacticalDefinition.p_tremolo_type__Chr58__bare_unsigned(p)
    tremolo_type : ':' bare_unsigned

```

### Special methods

```

(AbjadObject).__eq__(expr)
    True when ID of expr equals ID of Abjad object.

    Returns boolean.

(AbjadObject).__ne__(expr)
    True when ID of expr does not equal ID of Abjad object.

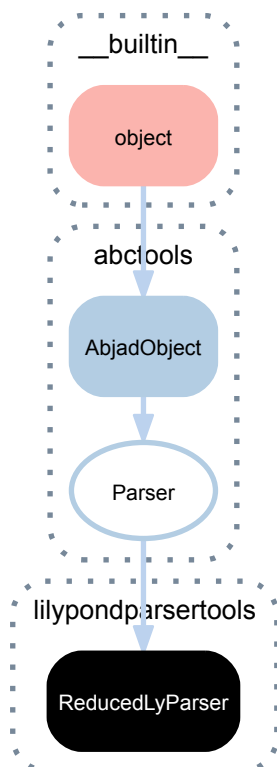
    Returns boolean.

(AbjadObject).__repr__()
    Interpreter representation of Abjad object.

    Returns string.

```

#### 59.2.10 lilypondparsertools.ReducedLyParser



```

class lilypondparsertools.ReducedLyParser(debug=False)
    Parses the “reduced-ly” syntax, a modified subset of LilyPond syntax:

```

```
>>> parser = lilypondparsertools.ReducedLyParser()
```

Understands LilyPond-like representation of notes, chords and rests:

```
>>> string = "c'4 r8. <b d' fs'>16"
>>> result = parser(string)
>>> f(result)
{
    c'4
    r8.
    <b d' fs'>16
}
```

Also parses bare duration as notes on middle-C, and negative bare durations as rests:

```
>>> string = '4 -8 16. -32'
>>> result = parser(string)
>>> f(result)
{
    c'4
    r8
    c'16.
    r32
}
```

Note that the leaf syntax is greedy, and therefore duration specifiers following pitch specifiers will be treated as part of the same expression. The following produces 2 leaves, rather than 3:

```
>>> string = "4 d' 4"
>>> result = parser(string)
>>> f(result)
{
    c'4
    d'4
}
```

Understands LilyPond-like default durations:

```
>>> string = "c'4 d' e' f'"
>>> result = parser(string)
>>> f(result)
{
    c'4
    d'4
    e'4
    f'4
}
```

Also understands various types of container specifications.

Can create arbitrarily nested tuplets:

```
>>> string = "2/3 { 4 4 3/5 { 8 8 8 } }"
>>> result = parser(string)
>>> f(result)
\times 2/3 {
  c'4
  c'4
  \tweak #'text #tuplet-number::calc-fraction-text
  \times 3/5 {
    c'8
    c'8
    c'8
  }
}
```

Can also create empty *FixedDurationContainers*:

```
>>> string = '{1/4} {3/4}'
>>> result = parser(string)
>>> for x in result: x
...
FixedDurationContainer(Duration(1, 4), [])
FixedDurationContainer(Duration(3, 4), [])
```

Can create measures too:

```
>>> string = '| 4/4 4 4 4 4 || 3/8 8 8 8 |'
>>> result = parser(string)
>>> for x in result: x
...
Measure(4/4, [c'4, c'4, c'4, c'4])
Measure(3/8, [c'8, c'8, c'8])
```

Finally, understands ties, slurs and beams:

```
>>> string = 'c16 [ ( d ~ d ) f ]'
>>> result = parser(string)
>>> f(result)
{
  c16 [ (
    d16 ~
    d16 )
    f16 ]
}
```

Return *ReducedLyParser* instance.

## Bases

- `abctools.Parser`
- `abctools.AbjadObject`
- `__builtin__.object`

## Read-only properties

(Parser) **.debug**  
True if the parser runs in debugging mode.

(Parser) **.lexer**  
The parser's PLY Lexer instance.

`ReducedLyParser.lexer_rules_object`

(Parser) **.logger**  
The parser's Logger instance.

(Parser) **.logger\_path**  
The output path for the parser's logfile.

(Parser) **.output\_path**  
The output path for files associated with the parser.

(Parser) **.parser**  
The parser's PLY LRParser instance.

`ReducedLyParser.parser_rules_object`

(Parser) **.pickle\_path**  
The output path for the parser's pickled parsing tables.

## Methods

`ReducedLyParser.p_apostrophes__APOSTROPHE` (*p*)  
apostrophes : APOSTROPHE

`ReducedLyParser.p_apostrophes__apostrophes__APOSTROPHE` (*p*)  
apostrophes : apostrophes APOSTROPHE

```

ReducedLyParser.p_beam__BRACKET_L(p)
    beam : BRACKET_L

ReducedLyParser.p_beam__BRACKET_R(p)
    beam : BRACKET_R

ReducedLyParser.p_chord_body__chord_pitches(p)
    chord_body : chord_pitches

ReducedLyParser.p_chord_body__chord_pitches__positive_leaf_duration(p)
    chord_body : chord_pitches positive_leaf_duration

ReducedLyParser.p_chord_pitches__CARAT_L__pitches__CARAT_R(p)
    chord_pitches : CARAT_L pitches CARAT_R

ReducedLyParser.p_commas__COMMA(p)
    commas : COMMA

ReducedLyParser.p_commas__commas__commas(p)
    commas : commas COMMA

ReducedLyParser.p_component__container(p)
    component : container

ReducedLyParser.p_component__fixed_duration_container(p)
    component : fixed_duration_container

ReducedLyParser.p_component__leaf(p)
    component : leaf

ReducedLyParser.p_component__tuplet(p)
    component : tuplet

ReducedLyParser.p_component_list__EMPTY(p)
    component_list :

ReducedLyParser.p_component_list__component_list__component(p)
    component_list : component_list component

ReducedLyParser.p_container__BRACE_L__component_list__BRACE_R(p)
    container : BRACE_L component_list BRACE_R

ReducedLyParser.p_dots__EMPTY(p)
    dots :

ReducedLyParser.p_dots__dots__DOT(p)
    dots : dots DOT

ReducedLyParser.p_error(p)

ReducedLyParser.p_fixed_duration_container__BRACE_L__FRACTION__BRACE_R(p)
    fixed_duration_container : BRACE_L FRACTION BRACE_R

ReducedLyParser.p_leaf__leaf_body__post_events(p)
    leaf : leaf_body post_events

ReducedLyParser.p_leaf_body__chord_body(p)
    leaf_body : chord_body

ReducedLyParser.p_leaf_body__note_body(p)
    leaf_body : note_body

ReducedLyParser.p_leaf_body__rest_body(p)
    leaf_body : rest_body

ReducedLyParser.p_measure__PIPE__FRACTION__component_list__PIPE(p)
    measure : PIPE FRACTION component_list PIPE

ReducedLyParser.p_negative_leaf_duration__INTEGER_N__dots(p)
    negative_leaf_duration : INTEGER_N dots

```



---

```

ReducedLyParser.p_note_body__pitch (p)
    note_body : pitch

ReducedLyParser.p_note_body__pitch__positive_leaf_duration (p)
    note_body : pitch positive_leaf_duration

ReducedLyParser.p_note_body__positive_leaf_duration (p)
    note_body : positive_leaf_duration

ReducedLyParser.p_pitch__PITCHNAME (p)
    pitch : PITCHNAME

ReducedLyParser.p_pitch__PITCHNAME__apostrophes (p)
    pitch : PITCHNAME apostrophes

ReducedLyParser.p_pitch__PITCHNAME__commas (p)
    pitch : PITCHNAME commas

ReducedLyParser.p_pitches__pitch (p)
    pitches : pitch

ReducedLyParser.p_pitches__pitches__pitch (p)
    pitches : pitches pitch

ReducedLyParser.p_positive_leaf_duration__INTEGER_P__dots (p)
    positive_leaf_duration : INTEGER_P dots

ReducedLyParser.p_post_event__beam (p)
    post_event : beam

ReducedLyParser.p_post_event__slur (p)
    post_event : slur

ReducedLyParser.p_post_event__tie (p)
    post_event : tie

ReducedLyParser.p_post_events__EMPTY (p)
    post_events :

ReducedLyParser.p_post_events__post_events__post_event (p)
    post_events : post_events post_event

ReducedLyParser.p_rest_body__RESTNAME (p)
    rest_body : RESTNAME

ReducedLyParser.p_rest_body__RESTNAME__positive_leaf_duration (p)
    rest_body : RESTNAME positive_leaf_duration

ReducedLyParser.p_rest_body__negative_leaf_duration (p)
    rest_body : negative_leaf_duration

ReducedLyParser.p_slur__PAREN_L (p)
    slur : PAREN_L

ReducedLyParser.p_slur__PAREN_R (p)
    slur : PAREN_R

ReducedLyParser.p_start__EMPTY (p)
    start :

ReducedLyParser.p_start__start__component (p)
    start : start component

ReducedLyParser.p_start__start__measure (p)
    start : start measure

ReducedLyParser.p_tie__TILDE (p)
    tie : TILDE

```

ReducedLyParser.**p\_tuplet\_FRACTION\_container** (*p*)  
 tuple : FRACTION container

ReducedLyParser.**t\_FRACTION** (*t*)  
 ([1-9]d\*/[1-9]d\*)

ReducedLyParser.**t\_INTEGER\_N** (*t*)  
 (-[1-9]d\*)

ReducedLyParser.**t\_INTEGER\_P** (*t*)  
 ([1-9]d\*)

ReducedLyParser.**t\_PITCHNAME** (*t*)  
 [a-g](fflsslflstqlftqlslqlf)?

ReducedLyParser.**t\_error** (*t*)

ReducedLyParser.**t\_newline** (*t*)  
 n+

(Parser) **.tokenize** (*input\_string*)  
 Tokenize *input\_string* and print results.

## Special methods

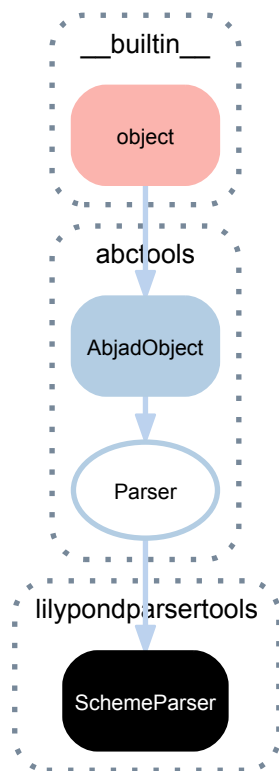
(Parser) **.\_\_call\_\_** (*input\_string*)  
 Parse *input\_string* and return result.

(AbjadObject) **.\_\_eq\_\_** (*expr*)  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_ne\_\_** (*expr*)  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

(AbjadObject) **.\_\_repr\_\_** ()  
 Interpreter representation of Abjad object.  
 Returns string.

### 59.2.11 lilypondparsertools.SchemeParser



**class** `lilypondparsertools.SchemeParser` (*debug=False*)  
*SchemeParser* mimics how LilyPond’s embedded Scheme parser behaves.

It parses a single Scheme expression and then stops, by raising a *SchemeParserFinishedException*.

The parsed expression and its exact length in characters are cached on the *SchemeParser* instance.

It is intended to be used only in conjunction with *LilyPondParser*.

Returns *SchemeParser* instance.

#### Bases

- `abctools.Parser`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Read-only properties

`(Parser).debug`  
 True if the parser runs in debugging mode.

`(Parser).lexer`  
 The parser’s PLY Lexer instance.

`SchemeParser.lexer_rules_object`

`(Parser).logger`  
 The parser’s Logger instance.

`(Parser).logger_path`  
 The output path for the parser’s logfile.

`(Parser).output_path`  
 The output path for files associated with the parser.

`(Parser).parser`  
 The parser's PLY LRParser instance.

`SchemeParser.parser_rules_object`

`(Parser).pickle_path`  
 The output path for the parser's pickled parsing tables.

## Methods

`SchemeParser.p_boolean__BOOLEAN(p)`  
 boolean : BOOLEAN

`SchemeParser.p_constant__boolean(p)`  
 constant : boolean

`SchemeParser.p_constant__number(p)`  
 constant : number

`SchemeParser.p_constant__string(p)`  
 constant : string

`SchemeParser.p_data__EMPTY(p)`  
 data :

`SchemeParser.p_data__data__datum(p)`  
 data : data datum

`SchemeParser.p_datum__constant(p)`  
 datum : constant

`SchemeParser.p_datum__list(p)`  
 datum : list

`SchemeParser.p_datum__symbol(p)`  
 datum : symbol

`SchemeParser.p_datum__vector(p)`  
 datum : vector

`SchemeParser.p_error(p)`

`SchemeParser.p_expression__QUOTE__datum(p)`  
 expression : QUOTE datum

`SchemeParser.p_expression__constant(p)`  
 expression : constant

`SchemeParser.p_expression__variable(p)`  
 expression : variable

`SchemeParser.p_form__expression(p)`  
 form : expression

`SchemeParser.p_forms__EMPTY(p)`  
 forms :

`SchemeParser.p_forms__forms__form(p)`  
 forms : forms form

`SchemeParser.p_list__L_PAREN__data__R_PAREN(p)`  
 list : L\_PAREN data R\_PAREN

`SchemeParser.p_list__L_PAREN__data__datum__PERIOD__datum__R_PAREN(p)`  
 list : L\_PAREN data datum PERIOD datum R\_PAREN

```

SchemeParser.p_number__DECIMAL (p)
    number : DECIMAL

SchemeParser.p_number__HEXADECIMAL (p)
    number : HEXADECIMAL

SchemeParser.p_number__INTEGER (p)
    number : INTEGER

SchemeParser.p_program__forms (p)
    program : forms

SchemeParser.p_string__STRING (p)
    string : STRING

SchemeParser.p_symbol__IDENTIFIER (p)
    symbol : IDENTIFIER

SchemeParser.p_variable__IDENTIFIER (p)
    variable : IDENTIFIER

SchemeParser.p_vector__HASH__L_PAREN__data__R_PAREN (p)
    vector : HASH L_PAREN data R_PAREN

SchemeParser.t_BOOLEAN (t)
    #(T|F|t|f)

SchemeParser.t_DECIMAL (t)
    (((-?[0-9]+).[0-9]*)|(-?[0-9]+))

SchemeParser.t_HASH (t)
    #

SchemeParser.t_HEXADECIMAL (t)
    (X|x)[A-Fa-f0-9]+

SchemeParser.t_IDENTIFIER (t)
    ([A-Za-z!$%&*/<>?~_^:=.+~*|+~]|...)

SchemeParser.t_INTEGER (t)
    (-?[0-9]+)

SchemeParser.t_L_PAREN (t)
    (

SchemeParser.t_R_PAREN (t)
    )

SchemeParser.t_anything (t)
    .

SchemeParser.t_error (t)

SchemeParser.t_newline (t)
    n+

SchemeParser.t_quote (t)
    “

SchemeParser.t_quote_440 (t)
    \[nt\””]

SchemeParser.t_quote_443 (t)
    [^\”“]+

SchemeParser.t_quote_446 (t)
    “

SchemeParser.t_quote_456 (t)
    .

```

```
SchemeParser.t_quote_error (t)
SchemeParser.t_whitespace (t)
  [ tr]+
(Parser).tokenize (input_string)
  Tokenize input string and print results.
```

## Special methods

```
(Parser).__call__ (input_string)
  Parse input_string and return result.

(ObjadObject).__eq__ (expr)
  True when ID of expr equals ID of Abjad object.

  Returns boolean.

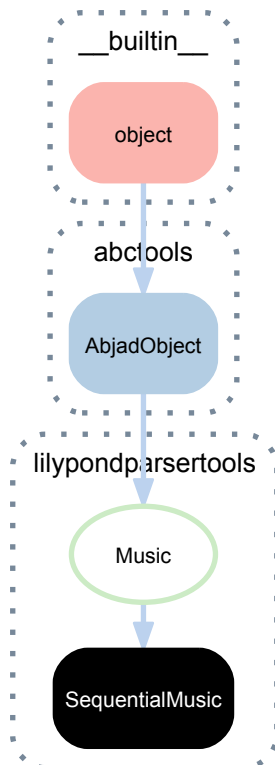
(ObjadObject).__ne__ (expr)
  True when ID of expr does not equal ID of Abjad object.

  Returns boolean.

(ObjadObject).__repr__ ()
  Interpreter representation of Abjad object.

  Returns string.
```

## 59.2.12 lilypondparsertools.SequentialMusic



```
class lilypondparsertools.SequentialMusic (music)
  Abjad model of the LilyPond AST sequential music node.
```

## Bases

- `lilypondparsertools.Music`
- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

`SequentialMusic.construct()`

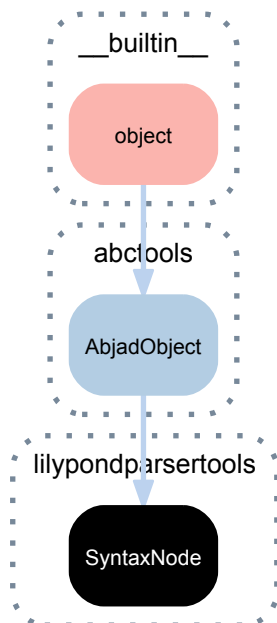
## Special methods

`(AbjadObject).__eq__(expr)`  
 True when ID of *expr* equals ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__ne__(expr)`  
 True when ID of *expr* does not equal ID of Abjad object.  
 Returns boolean.

`(AbjadObject).__repr__()`  
 Interpreter representation of Abjad object.  
 Returns string.

### 59.2.13 lilypondparsertools.SyntaxNode



**class** `lilypondparsertools.SyntaxNode` (*type*, *value*=[])  
 A node in an abstract syntax tree (AST).  
 Not composer-safe.  
 Used internally by LilyPondParser.

## Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Special methods

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`SyntaxNode.__getitem__(item)`

`SyntaxNode.__len__()`

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.  
Returns boolean.

`SyntaxNode.__repr__()`

`SyntaxNode.__str__()`

## 59.3 Functions

### 59.3.1 `lilypondparsertools.parse_reduced_ly_syntax`

`lilypondparsertools.parse_reduced_ly_syntax(string)`  
Parse the reduced LilyPond rhythmic syntax:

```
>>> string = '4 -4. 8.. 5/3 { } 4'
>>> result = lilypondparsertools.parse_reduced_ly_syntax(string)
```

```
>>> for x in result:
...     x
...
Note("c'4")
Rest('r4.')
Note("c'8..")
Tuplet(5/3, [])
Note("c'4")
```

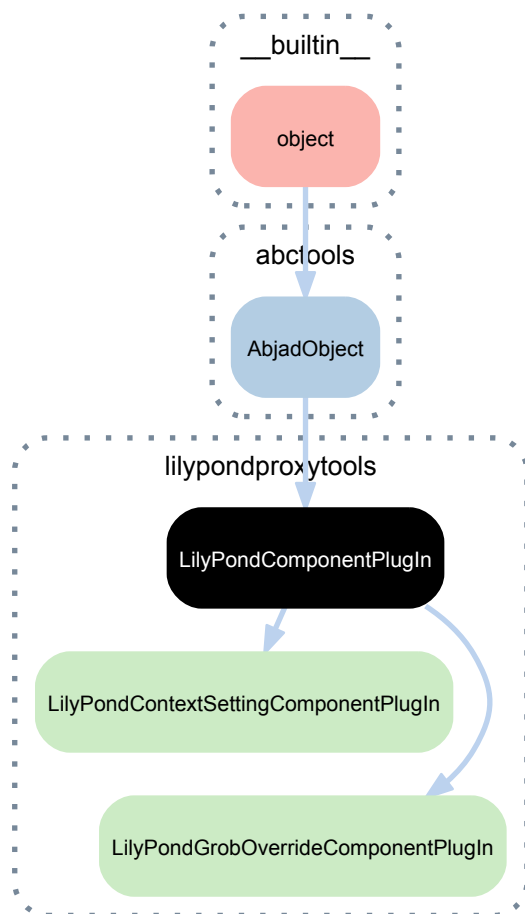
Returns list.



# LILYPONDPROXYTOOLS

## 60.1 Concrete classes

### 60.1.1 lilypondproxytools.LilyPondComponentPlugIn



**class** `lilypondproxytools.LilyPondComponentPlugIn` (*\*\*kwargs*)  
Shared LilyPond grob proxy and LilyPond context proxy functionality.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

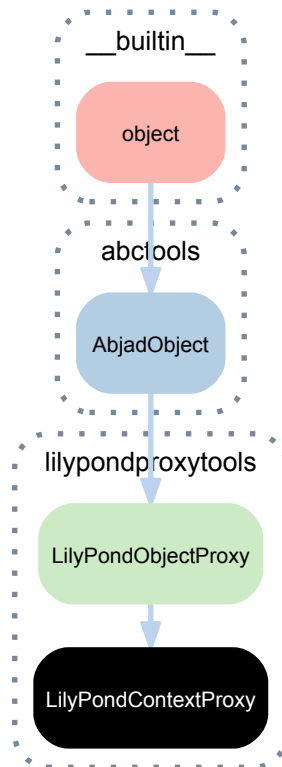
## Special methods

`LilyPondComponentPlugIn.__eq__(arg)`

`LilyPondComponentPlugIn.__ne__(arg)`

`LilyPondComponentPlugIn.__repr__()`

### 60.1.2 lilypondproxytools.LilyPondContextProxy



```
class lilypondproxytools.LilyPondContextProxy (**kwargs)
    LilyPond context proxy.
```

## Bases

- `lilypondproxytools.LilyPondObjectProxy`
- `abctools.AbjadObject`
- `__builtin__.object`

## Special methods

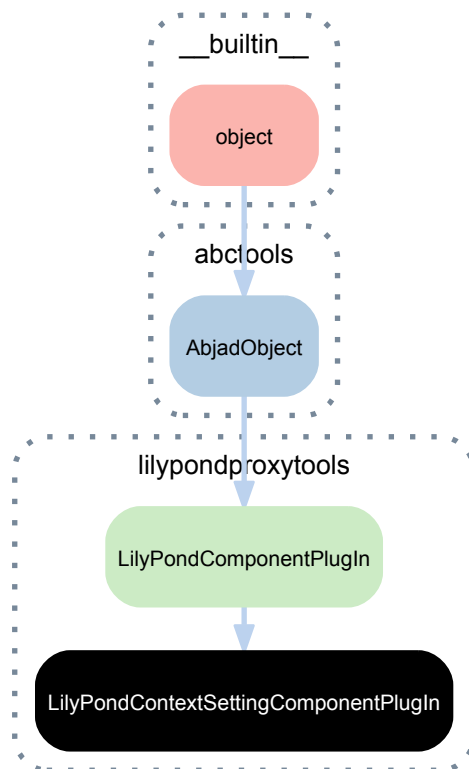
`(LilyPondObjectProxy).__copy__()`

`(LilyPondObjectProxy).__eq__(arg)`

`(LilyPondObjectProxy).__ne__(arg)`

`(LilyPondObjectProxy).__repr__()`

### 60.1.3 lilypondproxytools.LilyPondContextSettingComponentPlugIn



**class** `lilypondproxytools.LilyPondContextSettingComponentPlugIn` (*\*\*kwargs*)  
 LilyPond context setting namespace.

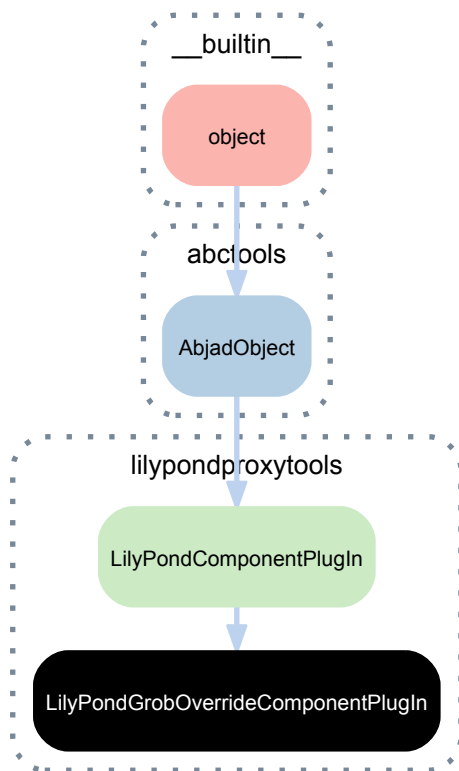
#### Bases

- `lilypondproxytools.LilyPondComponentPlugIn`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`(LilyPondComponentPlugIn).__eq__(arg)`  
`LilyPondContextSettingComponentPlugIn.__getattr__(name)`  
`(LilyPondComponentPlugIn).__ne__(arg)`  
`LilyPondContextSettingComponentPlugIn.__repr__()`

### 60.1.4 lilypondproxytools.LilyPondGrobOverrideComponentPlugIn



```
class lilypondproxytools.LilyPondGrobOverrideComponentPlugIn (**kwargs)
    LilyPond grob override component plug-in.
```

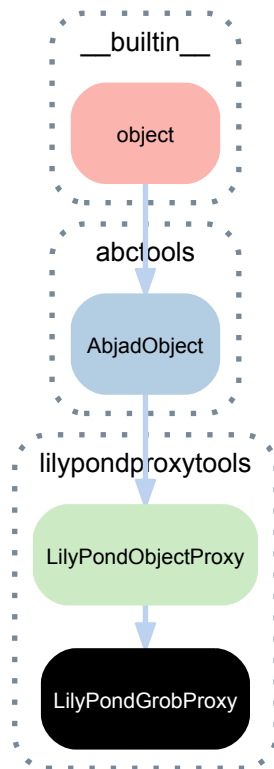
#### Bases

- `lilypondproxytools.LilyPondComponentPlugIn`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

```
(LilyPondComponentPlugIn).__eq__(arg)
LilyPondGrobOverrideComponentPlugIn.__getattr__(name)
(LilyPondComponentPlugIn).__ne__(arg)
LilyPondGrobOverrideComponentPlugIn.__repr__()
LilyPondGrobOverrideComponentPlugIn.__setattr__(attr, value)
```

### 60.1.5 lilypondproxytools.LilyPondGrobProxy



```
class lilypondproxytools.LilyPondGrobProxy(**kwargs)
    LilyPond grob proxy.
```

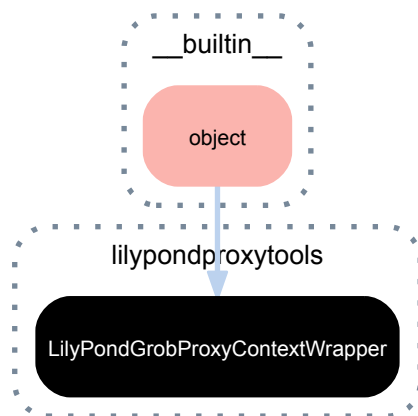
#### Bases

- `lilypondproxytools.LilyPondObjectProxy`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

```
LilyPondGrobProxy.__copy__()
(LilyPondObjectProxy).__eq__(arg)
(LilyPondObjectProxy).__ne__(arg)
(LilyPondObjectProxy).__repr__()
```

### 60.1.6 lilypondproxytools.LilyPondGrobProxyContextWrapper



**class** `lilypondproxytools.LilyPondGrobProxyContextWrapper`  
Context wrapper for LilyPond grob overrides.

#### Bases

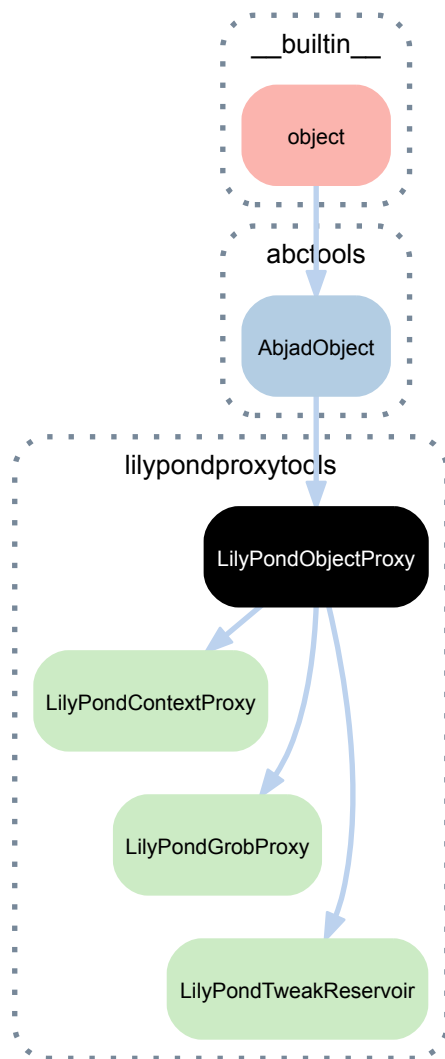
- `__builtin__.object`

#### Special methods

`LilyPondGrobProxyContextWrapper.__getattr__` (*name*)

`LilyPondGrobProxyContextWrapper.__repr__` ()

### 60.1.7 lilypondproxytools.LilyPondObjectProxy



**class** `lilypondproxytools.LilyPondObjectProxy` (*\*\*kwargs*)  
 Shared LilyPond grob proxy and LilyPond context proxy functionality.

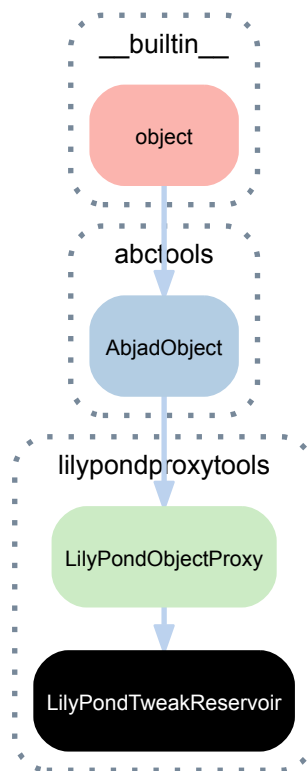
#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`LilyPondObjectProxy.__copy__()`  
`LilyPondObjectProxy.__eq__(arg)`  
`LilyPondObjectProxy.__ne__(arg)`  
`LilyPondObjectProxy.__repr__()`

### 60.1.8 lilypondproxytools.LilyPondTweakReservoir



```
class lilypondproxytools.LilyPondTweakReservoir (**kwargs)
    LilyPond tweak reservoir.
```

#### Bases

- `lilypondproxytools.LilyPondObjectProxy`
- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

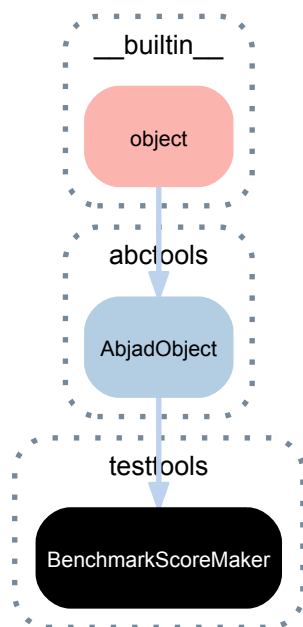
```
(LilyPondObjectProxy).__copy__()
(LilyPondObjectProxy).__eq__(arg)
(LilyPondObjectProxy).__ne__(arg)
(LilyPondObjectProxy).__repr__()
```



# TESTTOOLS

## 61.1 Concrete classes

### 61.1.1 testtools.BenchmarkScoreMaker



**class testtools.BenchmarkScoreMaker**  
Benchmark score maker:

```
>>> benchmark_score_maker = testtools.BenchmarkScoreMaker()
```

```
>>> benchmark_score_maker  
BenchmarkScoreMaker()
```

Use to instantiate scores for benchmark testing.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

## Methods

`BenchmarkScoreMaker.make_bound_hairpin_score_01()`

Make 200-note voice with p-to-f bound crescendo spanner on every 4 notes.

2.12 (r9726) initialization:	279,448 function calls
------------------------------	------------------------

2.12 (r9726) LilyPond format:	124,517 function calls
-------------------------------	------------------------

`BenchmarkScoreMaker.make_bound_hairpin_score_02()`

Make 200-note voice with p-to-f bound crescendo spanner on every 20 notes.

2.12 (r9726) initialization:	268,845 function calls
------------------------------	------------------------

2.12 (r9726) LilyPond format:	117,846 function calls
-------------------------------	------------------------

`BenchmarkScoreMaker.make_bound_hairpin_score_03()`

Make 200-note voice with p-to-f bound crescendo spanner on every 100 notes.

2.12 (r9726) initialization:	267,417 function calls
------------------------------	------------------------

2.12 (r9726) LilyPond format:	116,534 function calls
-------------------------------	------------------------

`BenchmarkScoreMaker.make_context_mark_score_01()`

Make 200-note voice with dynamic mark on every 20th note:

2.12 (r9704) initialization:	630,433 function calls
------------------------------	------------------------

2.12 (r9710) initialization:	235,120 function calls
------------------------------	------------------------

2.12 r(9726) initialization:	235,126 function calls
------------------------------	------------------------

2.12 (r9704) LilyPond format:	136,637 function calls
-------------------------------	------------------------

2.12 (r9710) LilyPond format:	82,730 function calls
-------------------------------	-----------------------

2.12 (r9726) LilyPond format:	88,382 function calls
-------------------------------	-----------------------

`BenchmarkScoreMaker.make_context_mark_score_02()`

Make 200-note staff with dynamic mark on every 4th note.

2.12 (r9704) initialization:	4,632,761 function calls
------------------------------	--------------------------

2.12 (r9710) initialization:	327,280 function calls
------------------------------	------------------------

2.12 (r9726) initialization:	325,371 function calls
------------------------------	------------------------

2.12 (r9704) LilyPond format:	220,277 function calls
-------------------------------	------------------------

2.12 (r9710) LilyPond format:	84,530 function calls
-------------------------------	-----------------------

2.12 (r9726) LilyPond format:	90,056 function calls
-------------------------------	-----------------------

`BenchmarkScoreMaker.make_context_mark_score_03()`

Make 200-note staff with dynamic mark on every note.

2.12 (r9704) initialization:	53,450,195 function calls (!!)
------------------------------	--------------------------------

2.12 (r9710) initialization:	2,124,500 function calls
------------------------------	--------------------------

2.12 (r9724) initialization:	2,122,591 function calls
------------------------------	--------------------------

2.12 (r9704) LilyPond format:	533,927 function calls
-------------------------------	------------------------

2.12 (r9710) LilyPond format:	91,280 function calls
-------------------------------	-----------------------

2.12 (r9724) LilyPond format:	96,806 function calls
-------------------------------	-----------------------

`BenchmarkScoreMaker.make_hairpin_score_01()`

Make 200-note voice with crescendo spanner on every 4 notes.

2.12 (r9726) initialization:	248,502 function calls
------------------------------	------------------------

2.12 (r9728) initialization:	248,502 function calls
------------------------------	------------------------

2.12 (r9726) LilyPond format:	138,313 function calls
-------------------------------	------------------------

2.12 (r9728) LilyPond format:	134,563 function calls
-------------------------------	------------------------

BenchmarkScoreMaker.**make\_hairpin\_score\_02**()

Make 200-note voice with crescendo spanner on every 20 notes.

2.12	(r9726)	initialization:	248,687	function calls
2.12	(r9728)	initialization:	248,687	function calls
2.12	(r9726)	LilyPond format:	134,586	function calls
2.12	(r9728)	LilyPond format:	129,836	function calls

BenchmarkScoreMaker.**make\_hairpin\_score\_03**()

Make 200-note voice with crescendo spanner on every 100 notes.

2.12	(r9726)	initialization:	249,363	function calls
2.12	(r9726)	initialization:	249,363	function calls
2.12	(r9726)	LilyPond format:	133,898	function calls
2.12	(r9728)	LilyPond format:	128,948	function calls

BenchmarkScoreMaker.**make\_score\_00**()

Make 200-note voice (with nothing else).

2.12	(r9710)	initialization:	156,821	function calls
2.12	(r9726)	initialization:	156,827	function calls
2.12	(r9703)	LilyPond format:	99,127	function calls
2.12	(r9710)	LilyPond format:	100,126	function calls
2.12	(r9726)	LilyPond format:	105,778	function calls

BenchmarkScoreMaker.**make\_spanner\_score\_01**()

Make 200-note voice with durated complex beam spanner on every 4 notes.

2.12	(r9710)	initialization:	248,654	function calls
2.12	(r9724)	initialization:	248,660	function calls
2.12	(r9703)	LilyPond format:	425,848	function calls
2.12	(r9710)	LilyPond format:	426,652	function calls
2.12	(r9724)	LilyPond format:	441,884	function calls

BenchmarkScoreMaker.**make\_spanner\_score\_02**()

Make 200-note voice with durated complex beam spanner on every 20 notes.

2.12	(r9710)	initialization:	250,954	function calls
2.12	(r9724)	initialization:	248,717	function calls
2.12	(r9703)	LilyPond format:	495,768	function calls
2.12	(r9710)	LilyPond format:	496,572	function calls
2.12	(r9724)	LilyPond format:	511,471	function calls

BenchmarkScoreMaker.**make\_spanner\_score\_03**()

Make 200-note voice with durated complex beam spanner on every 100 notes.

2.12	(r9710)	initialization:	251,606	function calls
2.12	(r9724)	initialization:	249,369	function calls
2.12	(r9703)	LilyPond format:	509,752	function calls
2.12	(r9710)	LilyPond format:	510,556	function calls
2.12	(r9724)	LilyPond format:	525,463	function calls

BenchmarkScoreMaker.**make\_spanner\_score\_04**()

Make 200-note voice with slur spanner on every 4 notes.

2.12	(r9724)	initialization:	245,683	function calls
2.12	(r9703)	LilyPond format:	125,577	function calls
2.12	(r9724)	LilyPond format:	111,341	function calls

BenchmarkScoreMaker.**make\_spanner\_score\_05**()

Make 200-note voice with slur spanner on every 20 notes.

2.12 (r9724) initialization:	248,567 function calls
2.12 (r9703) LilyPond format:	122,177 function calls
2.12 (r9724) LilyPond format:	107,486 function calls

BenchmarkScoreMaker.**make\_spanner\_score\_06**()

Make 200-note voice with slur spanner on every 100 notes.

2.12 (r9724) initialization:	249,339 function calls
2.12 (r9703) LilyPond format:	121,497 function calls
2.12 (r9724) LilyPond format:	106,718 function calls

BenchmarkScoreMaker.**make\_spanner\_score\_07**()

Make 200-note voice with (vanilla) beam spanner on every 4 notes.

2.12 (r9724) initialization:	245,683 function calls
2.12 (r9703) LilyPond format:	125,577 function calls
2.12 (r9724) LilyPond format:	132,556 function calls

BenchmarkScoreMaker.**make\_spanner\_score\_08**()

Make 200-note voice with (vanilla) beam spanner on every 20 notes.

2.12 (r9724) initialization:	248,567 function calls
2.12 (r9703) LilyPond format:	122,177 function calls
2.12 (r9724) LilyPond format:	129,166 function calls

BenchmarkScoreMaker.**make\_spanner\_score\_09**()

Make 200-note voice with (vanilla) beam spanner on every 100 notes.

2.12 (r9724) initialization:	249,339 function calls
2.12 (r9703) LilyPond format:	121,497 function calls
2.12 (r9724) LilyPond format:	128,494 function calls

## Special methods

(AbjadObject).**\_\_eq\_\_**(*expr*)

True when ID of *expr* equals ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_ne\_\_**(*expr*)

True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject).**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.

## 61.2 Functions

### 61.2.1 `testtools.apply_additional_layout`

`testtools.apply_additional_layout` (*lilypond\_file*)

Configure multiple-voice rhythmic staves in *lilypond\_file*.

Operates in place and returns none.

### 61.2.2 `testtools.compare`

`testtools.compare` (*string\_1*, *string\_2*)

### 61.2.3 `testtools.read_test_output`

`testtools.read_test_output` (*full\_file\_name*, *current\_function\_name*)

Read test output.

### 61.2.4 `testtools.write_test_output`

`testtools.write_test_output` (*output*, *full\_file\_name*, *test\_function\_name*, *cache\_ly=False*,  
*cache\_pdf=False*, *go=False*, *render\_pdf=False*)

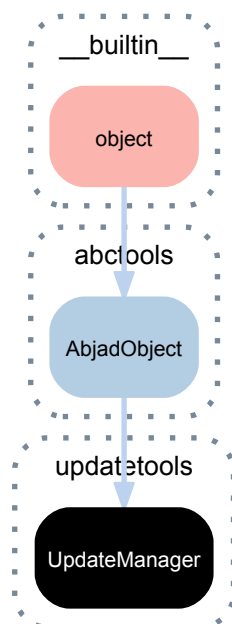
Write test output.



# UPDATETOOLS

## 62.1 Concrete classes

### 62.1.1 updatetools.UpdateManager



**class** `updatetools.UpdateManager`  
Update start offset, stop offsets and marks everywhere in score.

#### Bases

- `abctools.AbjadObject`
- `__builtin__.object`

#### Special methods

`(AbjadObject).__eq__(expr)`  
True when ID of *expr* equals ID of Abjad object.  
Returns boolean.

`(AbjadObject).__ne__(expr)`  
True when ID of *expr* does not equal ID of Abjad object.

Returns boolean.

(AbjadObject) .**\_\_repr\_\_**()

Interpreter representation of Abjad object.

Returns string.



# INDEX

## A

- AbjadAPIGenerator (class in abjad.tools.sequencetools.all\_are\_equal() (in module abjad.tools.sequencetools.all\_are\_equal), 1531, 893)
- AbjadBookProcessor (class in abjad.tools.sequencetools.all\_are\_integer\_equivalent\_exprs() (in module abjad.tools.abjadbooktools.AbjadBookProcessor.AbjadBookProcessor), 1390, 893)
- AbjadBookScript (class in abjad.tools.sequencetools.all\_are\_integer\_equivalent\_numbers() (in module abjad.tools.abjadbooktools.AbjadBookScript.AbjadBookScript), 1392, 894)
- AbjadConfiguration (class in abjad.tools.sequencetools.all\_are\_nonnegative\_integer\_equivalent\_numbers() (in module abjad.tools.configurationtools.AbjadConfiguration.AbjadConfiguration), 1401, 894)
- AbjadObject (class in abjad.tools.abctools.AbjadObject.AbjadObject), all\_are\_nonnegative\_integer\_powers\_of\_two() (in module abjad.tools.sequencetools.all\_are\_nonnegative\_integer\_powers\_of\_two), 1387, 894)
- AbjadRevisionToken (class in abjad.tools.lilypondfiletools.AbjadRevisionToken.AbjadRevisionToken), all\_are\_nonnegative\_integers() (in module abjad.tools.sequencetools.all\_are\_nonnegative\_integers), 334, 894)
- AbjDevScript (class in abjad.tools.developerscripttools.AbjDevScript.AbjDevScript), all\_are\_numbers() (in module abjad.tools.sequencetools.all\_are\_numbers), 1483, 895)
- AbjGrepScript (class in abjad.tools.developerscripttools.AbjGrepScript.AbjGrepScript), all\_are\_pairs() (in module abjad.tools.sequencetools.all\_are\_pairs), 1485, 895)
- AbjUpScript (class in abjad.tools.developerscripttools.AbjUpScript.AbjUpScript), all\_are\_pairs\_of\_types() (in module abjad.tools.sequencetools.all\_are\_pairs\_of\_types), 1487, 895)
- Accidental (class in abjad.tools.pitchtools.Accidental.Accidental), all\_are\_positive\_integer\_equivalent\_numbers() (in module abjad.tools.sequencetools.all\_are\_positive\_integer\_equivalent\_numbers), 516, 896)
- Accordion (class in abjad.tools.instrumenttools.Accordion.Accordion), all\_are\_positive\_integers() (in module abjad.tools.sequencetools.all\_are\_positive\_integers), 107, 896)
- add\_bell\_music\_to\_score() (in module abjad.demos.part.add\_bell\_music\_to\_score), all\_are\_unequal() (in module abjad.tools.sequencetools.all\_are\_unequal), 1378, 896)
- add\_string\_music\_to\_score() (in module abjad.demos.part.add\_string\_music\_to\_score), 1378, 896)
- add\_terminal\_newlines() (in module abjad.tools.stringtools.add\_terminal\_newlines), AltoFlute (class in abjad.tools.instrumenttools.AltoFlute.AltoFlute), 1121, 113)
- all\_are\_assignable\_integers() (in module abjad.tools.sequencetools.all\_are\_assignable\_integers), AltoSaxophone (class in abjad.tools.instrumenttools.AltoSaxophone.AltoSaxophone), 119

AltoTrombone	(class in ab-	1123	
	jad.tools.instrumenttools.AltoTrombone.AltoTrombone),		
124			
Annotation	(class in ab-	417	
	jad.tools.marktools.Annotation.Annotation),		
367			
Articulation	(class in ab-		
	jad.tools.marktools.Articulation.Articulation),		
append SpacerSkipToUnderfullMeasure()		369	
(in module ab-	AssignabilityError	(class in ab-	
jad.tools.measuretools.append SpacerSkipToUnderfullMeasure()),	jad.tools.exceptiontools.AssignabilityError),		
445		1653	
append SpacerSkipsToUnderfullMeasuresInExpr()	AttackPointOptimizer	(class in ab-	
(in module ab-	jad.tools.quantizationtools.AttackPointOptimizer.AttackPointO		
jad.tools.measuretools.append SpacerSkipsToUnderfullMeasuresInExpr()),			
445		641	
AttributedBlock	(class in ab-		
apply_accidental_to_named_pitch()	(in module ab-	jad.tools.lilypondfiletools.AttributedBlock.AttributedBlock),	
jad.tools.pitchtools.apply_accidental_to_named_pitch()),		329	
608			
AttributeInspectionAgent	(class in ab-		
apply_additional_layout()	(in module ab-	jad.tools.mutationtools.AttributeInspectionAgent.AttributeInsp	
jad.tools.testtools.apply_additional_layout()),		453	
1753			
apply_bowing_marks()	(in module ab-		
jad.demos.part.apply_bowing_marks),			
1378			
apply_dynamic_marks()	(in module ab-		
jad.demos.part.apply_dynamic_marks),			
1378			
apply_expressive_marks()	(in module ab-		
jad.demos.part.apply_expressive_marks),			
1378			
apply_final_bar_lines()	(in module ab-		
jad.demos.part.apply_final_bar_lines),			
1378			
apply_full_measure_tuplets_to_contents_of_measures_in_expr()		933	
(in module ab-	BaseResidueClass	(class in ab-	
jad.tools.measuretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr()),	jad.tools.sievetools.BaseResidueClass.BaseResidueClass),		
446			
apply_page_breaks()	(in module ab-		
jad.demos.part.apply_page_breaks),			
1378			
apply_rehearsal_marks()	(in module ab-		
jad.demos.part.apply_rehearsal_marks),			
1378			
are_relatively_prime()	(in module ab-		
jad.tools.mathtools.are_relatively_prime),			
417			
arg_to_bidirectional_direction_string()	(in module ab-		
jad.tools.stringtools.arg_to_bidirectional_direction_string()),			
1121			
arg_to_bidirectional_lilypond_symbol()	(in module ab-		
jad.tools.stringtools.arg_to_bidirectional_lilypond_symbol()),			
1121			
arg_to_tridirectional_direction_string()	(in module ab-		
jad.tools.stringtools.arg_to_tridirectional_direction_string()),			
1122			
arg_to_tridirectional_lilypond_symbol()	(in module ab-		
jad.tools.stringtools.arg_to_tridirectional_lilypond_symbol()),			
1122			
arg_to_tridirectional_ordinal_constant()	(in module ab-		
jad.tools.stringtools.arg_to_tridirectional_ordinal_constant()),			
		955	

## B

BaritoneSaxophone	(class in ab-		
	jad.tools.instrumenttools.BaritoneSaxophone.BaritoneSaxophone),		
133			
BaritoneVoice	(class in ab-		
	jad.tools.instrumenttools.BaritoneVoice.BaritoneVoice),		
136			
BarLine	(class in ab-		
	jad.tools.marktools.BarLine.BarLine),		
372			
BaseResidueClass	(class in ab-		
	jad.tools.sievetools.BaseResidueClass.BaseResidueClass),		
933			
BassClarinet	(class in ab-		
	jad.tools.instrumenttools.BassClarinet.BassClarinet),		
139			
BassFlute	(class in ab-		
	jad.tools.instrumenttools.BassFlute.BassFlute),		
142			
Bassoon	(class in ab-		
	jad.tools.instrumenttools.Bassoon.Bassoon),		
154			
BassSaxophone	(class in ab-		
	jad.tools.instrumenttools.BassSaxophone.BassSaxophone),		
145			
BassTrombone	(class in ab-		
	jad.tools.instrumenttools.BassTrombone.BassTrombone),		
148			
BassVoice	(class in ab-		
	jad.tools.instrumenttools.BassVoice.BassVoice),		
151			
BeamedQuarterNoteCheck	(class in ab-		
	jad.tools.wellformednesstools.BeamedQuarterNoteCheck.BeamedQuarterNoteCheck),		
1352			
BeamSpanner	(class in ab-		
	jad.tools.spannertools.BeamSpanner.BeamSpanner),		
955			
BeamwiseQS	(class in ab-		

jad.tools.quantizationtools.BeatwiseQSchema.BeatwiseQSchemaIndicator	(class in ab-	jad.tools.tonalanalysistools.ChordQualityIndicator.ChordQualityIndicator)
BeatwiseQSchemaItem	(class in ab-	jad.tools.quantizationtools.BeatwiseQSchemaItem.ChordQualityIndicator)
BeatwiseQTarget	(class in ab-	jad.tools.quantizationtools.BeatwiseQTarget.BeatwiseQTarget)
BenchmarkScoreMaker	(class in ab-	jad.tools.testtools.BenchmarkScoreMaker.BenchmarkScoreMaker)
BendAfter	(class in ab-	jad.tools.marktools.BendAfter.BendAfter)
BFlatClarinet	(class in ab-	jad.tools.instrumenttools.BFlatClarinet.BFlatClarinet)
binomial_coefficient()	(in module ab-	jad.tools.mathtools.binomial_coefficient)
BookBlock	(class in ab-	jad.tools.lilypondfiletools.BookBlock.BookBlock)
BookpartBlock	(class in ab-	jad.tools.lilypondfiletools.BookpartBlock.BookpartBlock)
BoundedObject	(class in ab-	jad.tools.mathtools.BoundedObject.BoundedObject)
BracketSpanner	(class in ab-	jad.tools.spannertools.BraceSpanner.BraceSpanner)
BreakPointFunction	(class in ab-	jad.tools.datastructuretools.BreakPointFunction.BreakPointFunction)
BuildApiScript	(class in ab-	jad.tools.developerscripttools.BuildApiScript.BuildApiScript)
BurnishedRhythmMaker	(class in ab-	jad.tools.rhythmmakertools.BurnishedRhythmMaker.BurnishedRhythmMaker)
<b>C</b>		
capitalize_string_start()	(in module ab-	jad.tools.stringtools.capitalize_string_start)
Cello	(class in abjad.tools.instrumenttools.Cello.Cello)	
Check	(class in ab-	jad.tools.wellformednesstools.Check.Check)
choose_mozart_measures()	(in module ab-	jad.demos.mozart.choose_mozart_measures)
Chord	(class in abjad.tools.chordtools.Chord.Chord)	
ChordClass	(class in ab-	jad.tools.tonalanalysistools.ChordClass.ChordClass)
ChordQualityIndicator	(class in ab-	jad.tools.tonalanalysistools.ChordQualityIndicator.ChordQualityIndicator)
ClassCrawler	(class in ab-	jad.tools.documentationtools.ClassCrawler.ClassCrawler)
ClassDocumenter	(class in ab-	jad.tools.documentationtools.ClassDocumenter.ClassDocumenter)
ClefMark	(class in ab-	jad.tools.contexttools.ClefMark.ClefMark)
ClefMarkInventory	(class in ab-	jad.tools.contexttools.ClefMarkInventory.ClefMarkInventory)
Cluster	(class in ab-	jad.tools.containertools.Cluster.Cluster)
CodeBlock	(class in ab-	jad.tools.abjadbooktools.CodeBlock.CodeBlock)
CollapsingGraceHandler	(class in ab-	jad.tools.quantizationtools.CollapsingGraceHandler.CollapsingGraceHandler)
color_chord_note_heads_in_expr_by_pitch_class_color_map()	(in module ab-	jad.tools.labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map)
color_contents_of_container()	(in module ab-	jad.tools.labeltools.color_contents_of_container)
color_leaf()	(in module ab-	jad.tools.labeltools.color_leaf)
color_leaves_in_expr()	(in module ab-	jad.tools.labeltools.color_leaves_in_expr)
color_measure()	(in module ab-	jad.tools.labeltools.color_measure)
color_measures_with_non_power_of_two_denominators_in_expr()	(in module ab-	jad.tools.labeltools.color_measures_with_non_power_of_two_denominators_in_expr)

color_note_head_by_numbered_pitch_class_color_map()	1383
(in module abjad.tools.labeltools.color_note_head_by_numbered_pitch_class_color_map),	302
combine_markup_commands()	397
(in module abjad.tools.markuptools.combine_markup_commands),	1693
compare()	1753
(in module abjad.tools.testtools.compare),	ContiguousSelection
(class in abjad.tools.selectiontools.ContiguousSelection.ContiguousSelection),	865
compare_images()	1651
(in module abjad.tools.documentationtools.compare_images),	Contrabass
(class in abjad.tools.instrumenttools.Contrabass.Contrabass),	163
ComplexBeamSpanner	967
(class in abjad.tools.spannertools.ComplexBeamSpanner.ComplexBeamSpanner),	(class in abjad.tools.instrumenttools.ContrabassClarinet.ContrabassClarinet),
ComplexGlissandoSpanner	974
(class in abjad.tools.spannertools.ComplexGlissandoSpanner.ComplexGlissandoSpanner),	(class in abjad.tools.instrumenttools.ContrabassFlute.ContrabassFlute),
Component	11
(class in abjad.tools.componenttools.Component.Component),	Contrabassoon
(class in abjad.tools.instrumenttools.Contrabassoon.Contrabassoon),	175
CompoundInequality	1188
(class in abjad.tools.timerelationtools.CompoundInequality.CompoundInequality),	(class in abjad.tools.instrumenttools.ContrabassSaxophone.ContrabassSaxophone),
concatenate_trees()	1183
(in module abjad.tools.timeintervaltools.concatenate_trees),	ContraltoVoice
(class in abjad.tools.instrumenttools.ContraltoVoice.ContraltoVoice),	178
ConcatenatingGraceHandler	641
(class in abjad.tools.quantizationtools.ConcatenatingGraceHandler.ConcatenatingGraceHandler),	module abjad.tools.iotools.count_function_calls),
Configuration	274
(class in abjad.tools.configurationtools.Configuration.Configuration),	count_length_two_runs_in_sequence()
(in module abjad.tools.sequencetools.count_length_two_runs_in_sequence),	896
configure_lilypond_file()	1373
(in module abjad.demos.ferneyhough.configure_lilypond_file),	CountLinewidthsScript
(class in abjad.tools.developerscripttools.CountLinewidthsScript.CountLinewidthsScript),	1493
configure_lilypond_file()	1378
(in module abjad.demos.part.configure_lilypond_file),	CountToolsScript
(class in abjad.tools.developerscripttools.CountToolsScript.CountToolsScript),	1495
configure_score()	1373
(in module abjad.demos.ferneyhough.configure_score),	create_pitch_contour_reservoir()
(in module abjad.demos.part.create_pitch_contour_reservoir),	1379
configure_score()	1378
(in module abjad.demos.part.configure_score),	CrescendoSpanner
(class in abjad.tools.spannertools.CrescendoSpanner.CrescendoSpanner),	980
Container	19
(class in abjad.tools.containertools.Container.Container),	cumulative_products()
(in module abjad.tools.mathtools.cumulative_products),	418
contains_subsegment()	609
(in module abjad.tools.pitchtools.contains_subsegment),	cumulative_signed_weights()
(in module abjad.tools.mathtools.cumulative_signed_weights),	418
Context	47
(class in abjad.tools.contexttools.Context.Context),	cumulative_sums()
(in module abjad.tools.mathtools.cumulative_sums),	418
ContextBlock	340
(class in abjad.tools.lilypondfiletools.ContextBlock.ContextBlock),	cumulative_sums_zero()
(in module abjad.tools.mathtools.cumulative_sums_zero),	418
ContextManager	418
(class in abjad.tools.abctools.ContextManager.ContextManager),	

[cumulative\\_sums\\_zero\\_pairwise\(\)](#) (in module `abjad.tools.mathtools.cumulative_sums_zero_pairwise`), [419](#)

[CyclicList](#) (class in `abjad.tools.datastructuretools.CyclicList.CyclicList`), [1416](#)

[CyclicMatrix](#) (class in `abjad.tools.datastructuretools.CyclicMatrix.CyclicMatrix`), [1419](#)

[CyclicPayloadTree](#) (class in `abjad.tools.datastructuretools.CyclicPayloadTree.CyclicPayloadTree`), [1421](#)

[CyclicTuple](#) (class in `abjad.tools.datastructuretools.CyclicTuple.CyclicTuple`), [1434](#)

**D**

[divisors\(\)](#) (in module `abjad.tools.mathtools.divisors`), [419](#)

[DateTimeToken](#) (class in `abjad.tools.lilypondfiletools.DateTimeToken.DateTimeToken`), [343](#)

[DecrescendoSpanner](#) (class in `abjad.tools.spannertools.DecrescendoSpanner.DecrescendoSpanner`), [988](#)

[default\\_instrument\\_name\\_to\\_instrument\\_class\(\)](#) (in module `abjad.tools.instrumenttools.default_instrument_name_to_instrument_class`), [269](#)

[Descendants](#) (class in `abjad.tools.selectiontools.Descendants.Descendants`), [867](#)

[DeveloperScript](#) (class in `abjad.tools.developerscripttools.DeveloperScript.DeveloperScript`), [1479](#)

[difference\\_series\(\)](#) (in module `abjad.tools.mathtools.difference_series`), [419](#)

[DirectedMark](#) (class in `abjad.tools.marktools.DirectedMark.DirectedMark`), [365](#)

[DirectedSpanner](#) (class in `abjad.tools.spannertools.DirectedSpanner.DirectedSpanner`), [943](#)

[DirectoryScript](#) (class in `abjad.tools.developerscripttools.DirectoryScript.DirectoryScript`), [1481](#)

[DiscardingGraceHandler](#) (class in `abjad.tools.quantizationtools.DiscardingGraceHandler.DiscardingGraceHandler`), [642](#)

[DiscontiguousSpannerCheck](#) (class in `abjad.tools.wellformednesstools.DiscontiguousSpannerCheck.DiscontiguousSpannerCheck`), [1353](#)

[DistanceHeuristic](#) (class in `abjad.tools.quantizationtools.DistanceHeuristic.DistanceHeuristic`), [643](#)

[divide\\_number\\_by\\_ratio\(\)](#) (in module `abjad.tools.mathtools.divide_number_by_ratio`), [419](#)

[divide\\_sequence\\_elements\\_by\\_greatest\\_common\\_divisor\(\)](#) (in module `abjad.tools.sequencetools.divide_sequence_elements_by_greatest_common_divisor`), [897](#)

[DivisionBurnishedTaleaRhythmMaker](#) (class in `abjad.tools.rhythm makertools.DivisionBurnishedTaleaRhythmMaker.DivisionBurnishedTaleaRhythmMaker`), [725](#)

[DivisionIncisedNoteRhythmMaker](#) (class in `abjad.tools.rhythm makertools.DivisionIncisedNoteRhythmMaker.DivisionIncisedNoteRhythmMaker`), [730](#)

[DivisionIncisedRestRhythmMaker](#) (class in `abjad.tools.rhythm makertools.DivisionIncisedRestRhythmMaker.DivisionIncisedRestRhythmMaker`), [735](#)

[DivisionIncisedRhythmMaker](#) (class in `abjad.tools.rhythm makertools.DivisionIncisedRhythmMaker.DivisionIncisedRhythmMaker`), [716](#)

[Documenter](#) (class in `abjad.tools.documentationtools.Documenter.Documenter`), [1519](#)

[DuplicateIdCheck](#) (class in `abjad.tools.wellformednesstools.DuplicateIdCheck.DuplicateIdCheck`), [1354](#)

[durate\\_pitch\\_contour\\_reservoir\(\)](#) (in module `abjad.demos.part.durate_pitch_contour_reservoir`), [1379](#)

[DulatedComplexBeamSpanner](#) (class in `abjad.tools.spannertools.DulatedComplexBeamSpanner.DulatedComplexBeamSpanner`), [996](#)

[Duration](#) (class in `abjad.tools.durationtools.Duration.Duration`), [571](#)

[duration\\_and\\_possible\\_denominators\\_to\\_time\\_signature\(\)](#) (in module `abjad.tools.timesignaturetools.duration_and_possible_denominators_to_time_signature`), [1227](#)

[DynamicMark](#) (class in `abjad.tools.contexttools.DynamicMark.DynamicMark`), [55](#)

[DynamicTextSpanner](#) (class in `abjad.tools.spannertools.DynamicTextSpanner.DynamicTextSpanner`), [1004](#)

**E**

[edit\\_bass\\_voice\(\)](#) (in module `abjad.demos.part.edit_bass_voice`), [1379](#)

[edit\\_cello\\_voice\(\)](#) (in module `abjad.demos.part.edit_cello_voice`), [1379](#)

[edit\\_first\\_violin\\_voice\(\)](#) (in module `abjad.demos.part.edit_first_violin_voice`), [1379](#)

[edit\\_second\\_violin\\_voice\(\)](#) (in module `abjad.demos.part.edit_second_violin_voice`), [1379](#)

[edit\\_viola\\_voice\(\)](#) (in module `abjad.demos.part.edit_viola_voice`), [1379](#)

[EFlatClarinet](#) (class in `abjad.tools.instrumenttools.EFlatClarinet.EFlatClarinet`), [1379](#)

[181](#)  
[EmptyContainerCheck](#) (class in [abjad.tools.timesignaturetools](#).[EmptyContainerCheck](#).[EmptyContainerCheck](#)),  
[jad.tools.wellformednesstools](#).[EmptyContainerCheck](#).[EmptyContainerCheck](#)),  
[1355](#)  
[FixedDurationContainer](#) (class in [abjad.tools.containertools](#).[FixedDurationContainer](#).[FixedDurationContainer](#)),  
[EnglishHorn](#) (class in [abjad.tools.instrumenttools](#).[EnglishHorn](#).[EnglishHorn](#)),  
[25](#)  
[184](#)  
[FixedDurationTuplet](#) (class in [abjad.tools.tuplettools](#).[FixedDurationTuplet](#).[FixedDurationTuplet](#)),  
[EqualDivisionRhythmMaker](#) (class in [abjad.tools.rhythm makertools](#).[EqualDivisionRhythmMaker](#).[EqualDivisionRhythmMaker](#)),  
[jad.tools.rhythm makertools](#).[EqualDivisionRhythmMaker](#).[EqualDivisionRhythmMaker](#)),  
[739](#)  
[flatten\\_sequence\(\)](#) (in module [abjad.tools.sequencetools](#).[flatten\\_sequence](#)),  
[establish\\_metrical\\_hierarchy\(\)](#) (in module [abjad.tools.timesignaturetools](#).[establish\\_metrical\\_hierarchy](#)),  
[jad.tools.timesignaturetools](#).[establish\\_metrical\\_hierarchy](#)),  
[897](#)  
[1227](#)  
[flatten\\_sequence\\_at\\_indices\(\)](#) (in module [abjad.tools.sequencetools](#).[flatten\\_sequence\\_at\\_indices](#)),  
[EvenRunRhythmMaker](#) (class in [abjad.tools.rhythm makertools](#).[EvenRunRhythmMaker](#).[EvenRunRhythmMaker](#)),  
[jad.tools.rhythm makertools](#).[EvenRunRhythmMaker](#).[EvenRunRhythmMaker](#)),  
[742](#)  
[Flute](#) (class in [abjad.tools.instrumenttools](#).[Flute](#).[Flute](#)),  
[extend\\_measures\\_in\\_expr\\_and\\_apply\\_full\\_measure\\_tuplets\(\)](#) [187](#)  
[\(in module \[abjad.tools.measuretools\]\(#\).\[extend\\\_measures\\\_in\\\_expr\\\_and\\\_apply\\\_full\\\_measure\\\_tuplets\]\(#\)\),](#)  
[jad.tools.measuretools](#).[extend\\_measures\\_in\\_expr\\_and\\_apply\\_full\\_measure\\_tuplets](#)),  
[446](#)  
[1123](#)  
[format\\_input\\_lines\\_as\\_doc\\_string\(\)](#) (in module [abjad.tools.stringtools](#).[format\\_input\\_lines\\_as\\_doc\\_string](#)),  
[ExtentIndicator](#) (class in [abjad.tools.tonalanalysistools](#).[ExtentIndicator](#).[ExtentIndicator](#)),  
[jad.tools.tonalanalysistools](#).[ExtentIndicator](#).[ExtentIndicator](#)),  
[1292](#)  
[1124](#)  
[format\\_lilypond\\_attribute\(\)](#) (in module [abjad.tools.formattools](#).[format\\_lilypond\\_attribute](#)),  
[ExtraMarkError](#) (class in [abjad.tools.exceptiontools](#).[ExtraMarkError](#)),  
[jad.tools.exceptiontools](#).[ExtraMarkError](#)),  
[1654](#)  
[1683](#)  
[format\\_lilypond\\_value\(\)](#) (in module [abjad.tools.formattools](#).[format\\_lilypond\\_value](#)),  
[ExtraNamedComponentError](#) (class in [abjad.tools.exceptiontools](#).[ExtraNamedComponentError](#)),  
[jad.tools.exceptiontools](#).[ExtraNamedComponentError](#)),  
[1655](#)  
[1683](#)  
[fraction\\_to\\_proper\\_fraction\(\)](#) (in module [abjad.tools.mathtools](#).[fraction\\_to\\_proper\\_fraction](#)),  
[ExtraNoteHeadError](#) (class in [abjad.tools.exceptiontools](#).[ExtraNoteHeadError](#)),  
[jad.tools.exceptiontools](#).[ExtraNoteHeadError](#)),  
[1656](#)  
[420](#)  
[FrenchHorn](#) (class in [abjad.tools.instrumenttools](#).[FrenchHorn](#).[FrenchHorn](#)),  
[ExtraPitchError](#) (class in [abjad.tools.exceptiontools](#).[ExtraPitchError](#)),  
[jad.tools.exceptiontools](#).[ExtraPitchError](#)),  
[1657](#)  
[190](#)  
[FunctionCrawler](#) (class in [abjad.tools.documentationtools](#).[FunctionCrawler](#).[FunctionCrawler](#)),  
[ExtraSpannerError](#) (class in [abjad.tools.exceptiontools](#).[ExtraSpannerError](#)),  
[jad.tools.exceptiontools](#).[ExtraSpannerError](#)),  
[1658](#)  
[1536](#)  
**F**  
[FunctionDocumenter](#) (class in [abjad.tools.documentationtools](#).[FunctionDocumenter](#).[FunctionDocumenter](#)),  
[jad.tools.documentationtools](#).[FunctionDocumenter](#).[FunctionDocumenter](#)),  
[1538](#)  
[f\(\)](#) (in module [abjad.tools.iotools](#).[f](#)), [275](#)  
[factors\(\)](#) (in module [abjad.tools.mathtools](#).[factors](#)), [420](#)  
**G**  
[fill\\_measures\\_in\\_expr\\_with\\_full\\_measure\\_spacer\\_skips\(\)](#)  
[\(in module \[abjad.tools.measuretools\]\(#\).\[fill\\\_measures\\\_in\\\_expr\\\_with\\\_full\\\_measure\\\_spacer\\\_skips\]\(#\)\),](#)  
[jad.tools.measuretools](#).[fill\\_measures\\_in\\_expr\\_with\\_full\\_measure\\_spacer\\_skips](#)),  
[447](#)  
[1683](#)  
[get\\_all\\_format\\_contributions\(\)](#) (in module [abjad.tools.formattools](#).[get\\_all\\_format\\_contributions](#)),  
[fill\\_measures\\_in\\_expr\\_with\\_minimal\\_number\\_of\\_notes\(\)](#)  
[\(in module \[abjad.tools.measuretools\]\(#\).\[fill\\\_measures\\\_in\\\_expr\\\_with\\\_minimal\\\_number\\\_of\\\_notes\]\(#\)\),](#)  
[jad.tools.measuretools](#).[fill\\_measures\\_in\\_expr\\_with\\_minimal\\_number\\_of\\_notes](#)),  
[447](#)  
[148](#)  
[get\\_all\\_mark\\_format\\_contributions\(\)](#) (in module [abjad.tools.formattools](#).[get\\_all\\_mark\\_format\\_contributions](#)),  
[fill\\_measures\\_in\\_expr\\_with\\_repeated\\_notes\(\)](#)  
[\(in module \[abjad.tools.measuretools\]\(#\).\[fill\\\_measures\\\_in\\\_expr\\\_with\\\_repeated\\\_notes\]\(#\)\),](#)  
[jad.tools.measuretools](#).[fill\\_measures\\_in\\_expr\\_with\\_repeated\\_notes](#)),  
[447](#)  
[1683](#)  
[get\\_articulation\\_format\\_contributions\(\)](#) (in module [abjad.tools.formattools](#).[get\\_articulation\\_format\\_contributions](#)),  
[fill\\_measures\\_in\\_expr\\_with\\_time\\_signature\\_denominator\\_notes\(\)](#)  
[\(in module \[abjad.tools.measuretools\]\(#\).\[fill\\\_measures\\\_in\\\_expr\\\_with\\\_time\\\_signature\\\_denominator\\\_notes\]\(#\)\),](#)  
[jad.tools.measuretools](#).[fill\\_measures\\_in\\_expr\\_with\\_time\\_signature\\_denominator\\_notes](#)),  
[447](#)  
[1684](#)  
[get\\_comment\\_format\\_contributions\\_for\\_slot\(\)](#) (in module [abjad.tools.formattools](#).[get\\_comment\\_format\\_contributions\\_for\\_slot](#)),  
[fill\\_measures\\_in\\_expr\\_with\\_time\\_signature\\_denominator\\_notes\(\)](#)  
[\(in module \[abjad.tools.measuretools\]\(#\).\[fill\\\_measures\\\_in\\\_expr\\\_with\\\_time\\\_signature\\\_denominator\\\_notes\]\(#\)\),](#)  
[jad.tools.measuretools](#).[fill\\_measures\\_in\\_expr\\_with\\_time\\_signature\\_denominator\\_notes](#)),  
[447](#)  
[\(in module \[abjad.tools.measuretools\]\(#\).\[fill\\\_measures\\\_in\\\_expr\\\_with\\\_time\\\_signature\\\_denominator\\\_notes\]\(#\)\),](#)  
[jad.tools.measuretools](#).[fill\\_measures\\_in\\_expr\\_with\\_time\\_signature\\_denominator\\_notes](#)),  
[447](#)



[jad.tools.formattools.get\\_context\\_mark\\_format\\_contributions\\_for\\_slot\(\)](#), module [ab-](#)  
[1684](#) [jad.tools.measuretools.get\\_one\\_indexed\\_measure\\_number\\_in\\_expr\(\)](#)  
[get\\_context\\_mark\\_format\\_pieces\(\)](#) (in module [ab-](#) [448](#)  
[jad.tools.formattools.get\\_context\\_mark\\_format\\_pieces\(\)](#)  
[1684](#) [jad.tools.measuretools.get\\_previous\\_measure\\_from\\_component\(\)](#)  
[get\\_context\\_setting\\_format\\_contributions\(\)](#) (in module [ab-](#) [449](#)  
[jad.tools.formattools.get\\_context\\_setting\\_format\\_contributions\(\)](#)  
[1684](#) [jad.tools.measuretools.get\\_previous\\_measure\\_from\\_component\(\)](#)  
[get\\_current\\_function\\_name\(\)](#) (in module [ab-](#) [jad.tools.sequencetools.get\\_sequence\\_degree\\_of\\_rotational\\_symmetry\(\)](#)  
[jad.tools.introspectiontools.get\\_current\\_function\\_name\(\)](#), [898](#)  
[1689](#) [get\\_sequence\\_element\\_at\\_cyclic\\_index\(\)](#)  
[get\\_developer\\_script\\_classes\(\)](#) (in module [ab-](#) (in module [ab-](#)  
[jad.tools.developerscripttools.get\\_developer\\_script\\_classes\(\)](#), [898](#)  
[1518](#) [jad.tools.sequencetools.get\\_sequence\\_element\\_at\\_cyclic\\_index\(\)](#)  
[get\\_grob\\_override\\_format\\_contributions\(\)](#) [get\\_sequence\\_elements\\_at\\_indices\(\)](#) (in module [ab-](#)  
(a in module [ab-](#) [jad.tools.sequencetools.get\\_sequence\\_elements\\_at\\_indices\(\)](#),  
[jad.tools.formattools.get\\_grob\\_override\\_format\\_contributions\(\)](#),  
[1684](#) [get\\_sequence\\_elements\\_frequency\\_distribution\(\)](#)  
[get\\_grob\\_revert\\_format\\_contributions\(\)](#) (in module [ab-](#) (in module [ab-](#)  
[jad.tools.formattools.get\\_grob\\_revert\\_format\\_contributions\(\)](#), [899](#)  
[1684](#) [jad.tools.sequencetools.get\\_sequence\\_elements\\_frequency\\_distribution\(\)](#)  
[get\\_indices\\_of\\_sequence\\_elements\\_equal\\_to\\_true\(\)](#) [get\\_sequence\\_period\\_of\\_rotation\(\)](#) (in module [ab-](#)  
(a in module [ab-](#) [jad.tools.sequencetools.get\\_sequence\\_period\\_of\\_rotation\(\)](#),  
[jad.tools.sequencetools.get\\_indices\\_of\\_sequence\\_elements\\_equal\\_to\\_true\(\)](#),  
[898](#) [get\\_shared\\_numeric\\_sign\(\)](#) (in module [ab-](#)  
[get\\_last\\_output\\_file\\_name\(\)](#) (in module [ab-](#) [jad.tools.mathtools.get\\_shared\\_numeric\\_sign\(\)](#),  
[jad.tools.iotools.get\\_last\\_output\\_file\\_name\(\)](#), [421](#)  
[275](#) [get\\_spanner\\_format\\_contributions\(\)](#) (in module [ab-](#)  
[get\\_lilypond\\_command\\_mark\\_format\\_contributions\\_for\\_slot\(\)](#) [jad.tools.formattools.get\\_spanner\\_format\\_contributions\(\)](#),  
(a in module [ab-](#) [1685](#)  
[jad.tools.formattools.get\\_lilypond\\_command\\_mark\\_format\\_contributions\\_for\\_slot\(\)](#)  
[1684](#) (in module [ab-](#)  
[get\\_markup\\_format\\_contributions\(\)](#) (in module [ab-](#) [jad.tools.formattools.get\\_stem\\_tremolo\\_format\\_contributions\(\)](#),  
[jad.tools.formattools.get\\_markup\\_format\\_contributions\(\)](#), [1685](#)  
[1684](#) [GlissandoSpanner](#) (class in [ab-](#)  
[get\\_measure\\_that\\_starts\\_with\\_container\(\)](#) [jad.tools.spannertools.GlissandoSpanner.GlissandoSpanner\(\)](#),  
(a in module [ab-](#) [1010](#)  
[jad.tools.measuretools.get\\_measure\\_that\\_starts\\_with\\_container\(\)](#), (class in [ab-](#)  
[447](#) [jad.tools.instrumenttools.Glockenspiel.Glockenspiel\(\)](#),  
[get\\_measure\\_that\\_stops\\_with\\_container\(\)](#) [193](#)  
(a in module [ab-](#) [GraceContainer](#) (class in [ab-](#)  
[jad.tools.measuretools.get\\_measure\\_that\\_stops\\_with\\_container\(\)](#), [jad.tools.containertools.GraceContainer.GraceContainer\(\)](#),  
[448](#) [31](#)  
[get\\_named\\_pitch\\_from\\_pitch\\_carrier\(\)](#) (in module [ab-](#) [GraceHandler](#) (class in [ab-](#)  
[jad.tools.pitchtools.get\\_named\\_pitch\\_from\\_pitch\\_carrier\(\)](#), [jad.tools.quantizationtools.GraceHandler.GraceHandler\(\)](#),  
[609](#) [623](#)  
[get\\_next\\_measure\\_from\\_component\(\)](#) (in module [ab-](#) [GrandStaff](#) (class in [ab-](#)  
[jad.tools.measuretools.get\\_next\\_measure\\_from\\_component\(\)](#), [jad.tools.scoretools.GrandStaff.GrandStaff\(\)](#),  
[448](#) [823](#)  
[get\\_next\\_output\\_file\\_name\(\)](#) (in module [ab-](#) [graph\(\)](#) (in module [ab-](#) [jad.tools.iotools.graph\(\)](#), [275](#)  
[jad.tools.iotools.get\\_next\\_output\\_file\\_name\(\)](#), [GraphvizEdge](#) (class in [ab-](#)  
[275](#) [jad.tools.documentationtools.GraphvizEdge.GraphvizEdge\(\)](#),  
[get\\_numbered\\_pitch\\_class\\_from\\_pitch\\_carrier\(\)](#) [1540](#)  
(a in module [ab-](#) [GraphvizGraph](#) (class in [ab-](#)  
[jad.tools.pitchtools.get\\_numbered\\_pitch\\_class\\_from\\_pitch\\_carrier\(\)](#), [jad.tools.documentationtools.GraphvizGraph.GraphvizGraph\(\)](#),  
[610](#) [1541](#)  
[get\\_one\\_indexed\\_measure\\_number\\_in\\_expr\(\)](#) [GraphvizNode](#) (class in [ab-](#)

jad.tools.documentationtools.GraphvizNode.GraphvizNode),  
 1552  
 GraphvizObject (class in ab- ImportManager (class in ab-  
 jad.tools.documentationtools.GraphvizObject.GraphvizObject),  
 1521 jad.tools.importtools.ImportManager.ImportManager),  
 1555  
 GraphvizSubgraph (class in ab- ImpreciseTempoError (class in ab-  
 jad.tools.documentationtools.GraphvizSubgraph.GraphvizSubgraph),  
 1555 jad.tools.exceptiontools.ImpreciseTempoError),  
 greatest\_common\_divisor() (in module ab- IncisedRhythmMaker (class in ab-  
 jad.tools.mathtools.greatest\_common\_divisor), 719  
 421  
 greatest\_multiple\_less\_equal() (in module ab- increase\_sequence\_elements\_at\_indices\_by\_addenda()  
 jad.tools.mathtools.greatest\_multiple\_less\_equal), (in module ab-  
 421 jad.tools.sequencetools.increase\_sequence\_elements\_at\_indices  
 899  
 greatest\_power\_of\_two\_less\_equal() (in module ab- increase\_sequence\_elements\_cyclically\_by\_addenda()  
 jad.tools.mathtools.greatest\_power\_of\_two\_less\_equal), (in module ab-  
 422 jad.tools.sequencetools.increase\_sequence\_elements\_cyclically  
 GroupedRhythmicStavesScoreTemplate (class in ab- 900  
 jad.tools.scoretemplatetools.GroupedRhythmicStavesScoreTemplate),  
 815 infinity (class in abjad.tools.mathtools.infinity.infinity),  
 403  
 GroupedStavesScoreTemplate (class in ab- InheritanceGraph (class in ab-  
 jad.tools.scoretemplatetools.GroupedStavesScoreTemplate.GroupedStavesScoreTemplate),  
 817 jad.tools.documentationtools.InheritanceGraph.InheritanceGraph  
 1564  
 GuileProxy (class in ab- insert\_and\_transpose\_nested\_subruns\_in\_pitch\_class\_number\_list()  
 jad.tools.lilypondparsertools.GuileProxy.GuileProxy), (in module ab-  
 1694 jad.tools.pitchtools.insert\_and\_transpose\_nested\_subruns\_in\_pi  
 Guitar (class in ab- 610  
 jad.tools.instrumenttools.Guitar.Guitar), insert\_expr\_into\_lilypond\_file() (in module ab-  
 196 jad.tools.iotools.insert\_expr\_into\_lilypond\_file),  
 276  
**H**  
 instantiate\_pitch\_and\_interval\_test\_collection()  
 HairpinSpanner (class in ab- (in module ab-  
 jad.tools.spannertools.HairpinSpanner.HairpinSpanner), jad.tools.pitchtools.instantiate\_pitch\_and\_interval\_test\_collecti  
 1016 611  
 Harmonic (class in ab- Instrument (class in ab-  
 jad.tools.notetools.Harmonic.Harmonic), jad.tools.instrumenttools.Instrument.Instrument),  
 469 205  
 Harp (class in abjad.tools.instrumenttools.Harp.Harp), InstrumentationSpecifier (class in ab-  
 199 jad.tools.scoretools.InstrumentationSpecifier.InstrumentationSp  
 Harpsichord (class in ab- 830  
 jad.tools.instrumenttools.Harpsichord.Harpsichord), InstrumentError (class in ab-  
 202 jad.tools.exceptiontools.InstrumentError),  
 HeaderBlock (class in ab- 1660  
 jad.tools.lilypondfiletools.HeaderBlock.HeaderBlock), InstrumentInventory (class in ab-  
 344 jad.tools.instrumenttools.InstrumentInventory.InstrumentInvent  
 Heuristic (class in ab- 208  
 jad.tools.quantizationtools.Heuristic.Heuristic), integer\_equivalent\_number\_to\_integer() (in module ab-  
 624 jad.tools.mathtools.integer\_equivalent\_number\_to\_integer),  
 HiddenStaffSpanner (class in ab- 422  
 jad.tools.spannertools.HiddenStaffSpanner.HiddenStaffSpanner), integer\_to\_base\_k\_tuple() (in module ab-  
 1024 jad.tools.mathtools.integer\_to\_base\_k\_tuple),  
 HorizontalBracketSpanner (class in ab- 423  
 jad.tools.spannertools.HorizontalBracketSpanner.HorizontalBracketSpanner), integer\_to\_binary\_string() (in module ab-  
 1030 jad.tools.mathtools.integer\_to\_binary\_string),  
 HTMLOutputFormat (class in ab- 423  
 jad.tools.abjadbooktools.HTMLOutputFormat.HTMLOutputFormat), interlace\_sequences() (in module ab-  
 1394 jad.tools.sequencetools.interlace\_sequences),  
 900



IntermarkedHairpinCheck (class in abjad.tools.stringtools.is\_dash\_case\_string),  
 jad.tools.wellformednesstools.IntermarkedHairpinCheck, 1356  
 is\_dotted\_integer() (in module abjad.tools.mathtools.is\_dotted\_integer), 425  
 interpolate\_cosine() (in module abjad.tools.mathtools.interpolate\_cosine), 423  
 interpolate\_divide() (in module abjad.tools.mathtools.interpolate\_divide), 423  
 is\_formattable\_context\_mark\_for\_component() (in module abjad.tools.formattools.is\_formattable\_context\_mark\_for\_component), 1685  
 interpolate\_divide\_multiple() (in module abjad.tools.mathtools.interpolate\_divide\_multiple), 424  
 is\_fraction\_equivalent\_pair() (in module abjad.tools.sequencetools.is\_fraction\_equivalent\_pair), 900  
 interpolate\_exponential() (in module abjad.tools.mathtools.interpolate\_exponential), 424  
 is\_integer\_equivalent\_expr() (in module abjad.tools.mathtools.is\_integer\_equivalent\_expr), 426  
 interpolate\_linear() (in module abjad.tools.mathtools.interpolate\_linear), 424  
 is\_integer\_equivalent\_n\_tuple() (in module abjad.tools.sequencetools.is\_integer\_equivalent\_n\_tuple), 900  
 Interval (class in abjad.tools.pitchtools.Interval.Interval), 501  
 is\_integer\_equivalent\_number() (in module abjad.tools.mathtools.is\_integer\_equivalent\_number), 426  
 interval\_string\_to\_pair\_and\_indicators() (in module abjad.tools.mathtools.interval\_string\_to\_pair\_and\_indicators), 425  
 is\_integer\_equivalent\_pair() (in module abjad.tools.sequencetools.is\_integer\_equivalent\_pair), 901  
 IntervalClass (class in abjad.tools.pitchtools.IntervalClass.IntervalClass), 503  
 is\_integer\_equivalent\_singleton() (in module abjad.tools.sequencetools.is\_integer\_equivalent\_singleton), 901  
 IntervalClassSegment (class in abjad.tools.pitchtools.IntervalClassSegment.IntervalClassSegment), 519  
 is\_integer\_n\_tuple() (in module abjad.tools.sequencetools.is\_integer\_n\_tuple), 901  
 IntervalClassSet (class in abjad.tools.pitchtools.IntervalClassSet.IntervalClassSet), 522  
 is\_integer\_pair() (in module abjad.tools.sequencetools.is\_integer\_pair), 901  
 IntervalClassVector (class in abjad.tools.pitchtools.IntervalClassVector.IntervalClassVector), 525  
 is\_integer\_singleton() (in module abjad.tools.sequencetools.is\_integer\_singleton), 902  
 IntervalSegment (class in abjad.tools.pitchtools.IntervalSegment.IntervalSegment), 528  
 is\_lower\_camel\_case\_string() (in module abjad.tools.stringtools.is\_lower\_camel\_case\_string), 1125  
 IntervalSet (class in abjad.tools.pitchtools.IntervalSet.IntervalSet), 531  
 is\_monotonically\_decreasing\_sequence() (in module abjad.tools.sequencetools.is\_monotonically\_decreasing\_sequence), 902  
 IntervalVector (class in abjad.tools.pitchtools.IntervalVector.IntervalVector), 534  
 is\_monotonically\_increasing\_sequence() (in module abjad.tools.sequencetools.is\_monotonically\_increasing\_sequence), 902  
 inventory\_aggregate\_subsets() (in module abjad.tools.pitchtools.inventory\_aggregate\_subsets), 611  
 is\_n\_tuple() (in module abjad.tools.sequencetools.is\_n\_tuple), 903  
 InversionIndicator (class in abjad.tools.tonalanalysisistools.InversionIndicator.InversionIndicator), 1293  
 is\_negative\_integer() (in module abjad.tools.mathtools.is\_negative\_integer), 426  
 is\_assignable\_integer() (in module abjad.tools.mathtools.is\_assignable\_integer), 425  
 is\_nonnegative\_integer() (in module abjad.tools.mathtools.is\_nonnegative\_integer), 427  
 is\_dash\_case\_file\_name() (in module abjad.tools.stringtools.is\_dash\_case\_file\_name), 1124  
 is\_nonnegative\_integer\_equivalent\_number() (in module abjad.tools.mathtools.is\_nonnegative\_integer\_equivalent\_number), 902

<a href="#">427</a>	<a href="#">283</a>
<code>is_nonnegative_integer_power_of_two()</code>	<code>iterate_components_and_grace_containers_in_expr()</code>
(in module <code>abjad.tools.mathtools.is_nonnegative_integer_power_of_two</code> ),	(in module <code>abjad.tools.iterationtools.iterate_components_and_grace_containers_in_expr</code> ),
<a href="#">427</a>	<a href="#">283</a>
<code>is_null_tuple()</code> (in module <code>abjad.tools.sequencetools.is_null_tuple</code> ),	<code>iterate_components_depth_first()</code> (in module <code>abjad.tools.iterationtools.iterate_components_depth_first</code> ),
<a href="#">903</a>	<a href="#">284</a>
<code>is_pair()</code> (in module <code>abjad.tools.sequencetools.is_pair</code> ),	<code>iterate_components_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_components_in_expr</code> ),
<a href="#">903</a>	<a href="#">284</a>
<code>is_permutation()</code> (in module <code>abjad.tools.sequencetools.is_permutation</code> ),	<code>iterate_containers_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_containers_in_expr</code> ),
<a href="#">904</a>	<a href="#">284</a>
<code>is_positive_integer()</code> (in module <code>abjad.tools.mathtools.is_positive_integer</code> ),	<code>iterate_contexts_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_contexts_in_expr</code> ),
<a href="#">427</a>	<a href="#">285</a>
<code>is_positive_integer_equivalent_number()</code>	<code>iterate_leaf_pairs_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_leaf_pairs_in_expr</code> ),
(in module <code>abjad.tools.mathtools.is_positive_integer_equivalent_number</code> ),	<a href="#">285</a>
<a href="#">428</a>	<a href="#">285</a>
<code>is_positive_integer_power_of_two()</code> (in module <code>abjad.tools.mathtools.is_positive_integer_power_of_two</code> ),	<code>iterate_leaves_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_leaves_in_expr</code> ),
<a href="#">428</a>	<a href="#">285</a>
<code>is_repetition_free_sequence()</code> (in module <code>abjad.tools.sequencetools.is_repetition_free_sequence</code> ),	<code>iterate_logical_voice_from_component()</code>
<a href="#">904</a>	(in module <code>abjad.tools.iterationtools.iterate_logical_voice_from_component</code> ),
<code>is_restricted_growth_function()</code> (in module <code>abjad.tools.sequencetools.is_restricted_growth_function</code> ),	<a href="#">287</a>
<a href="#">904</a>	<code>iterate_logical_voice_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_logical_voice_in_expr</code> ),
<code>is_singleton()</code> (in module <code>abjad.tools.sequencetools.is_singleton</code> ),	<a href="#">288</a>
<a href="#">905</a>	<code>iterate_measures_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_measures_in_expr</code> ),
<code>is_snake_case_file_name()</code> (in module <code>abjad.tools.stringtools.is_snake_case_file_name</code> ),	<a href="#">289</a>
<a href="#">1125</a>	<code>iterate_named_pitch_pairs_in_expr()</code> (in module <code>abjad.tools.pitchtools.iterate_named_pitch_pairs_in_expr</code> ),
<code>is_snake_case_file_name_with_extension()</code>	<a href="#">612</a>
(in module <code>abjad.tools.stringtools.is_snake_case_file_name_with_extension</code> ),	<code>iterate_nontrivial_tie_chains_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_nontrivial_tie_chains_in_expr</code> ),
<a href="#">1125</a>	<a href="#">289</a>
<code>is_snake_case_package_name()</code> (in module <code>abjad.tools.stringtools.is_snake_case_package_name</code> ),	<code>iterate_notes_and_chords_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_notes_and_chords_in_expr</code> ),
<a href="#">1126</a>	<a href="#">290</a>
<code>is_snake_case_string()</code> (in module <code>abjad.tools.stringtools.is_snake_case_string</code> ),	<code>iterate_notes_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_notes_in_expr</code> ),
<a href="#">1126</a>	<a href="#">290</a>
<code>is_space_delimited_lowercase_string()</code> (in module <code>abjad.tools.stringtools.is_space_delimited_lowercase_string</code> ),	<code>iterate_out_of_range_notes_and_chords()</code>
<a href="#">1126</a>	(in module <code>abjad.tools.instrumenttools.iterate_out_of_range_notes_and_chords</code> ),
<code>is_strictly_decreasing_sequence()</code> (in module <code>abjad.tools.sequencetools.is_strictly_decreasing_sequence</code> ),	<a href="#">270</a>
<a href="#">905</a>	<code>iterate_pitched_tie_chains_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_pitched_tie_chains_in_expr</code> ),
<code>is_strictly_increasing_sequence()</code> (in module <code>abjad.tools.sequencetools.is_strictly_increasing_sequence</code> ),	<a href="#">292</a>
<a href="#">906</a>	<code>iterate_rests_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_rests_in_expr</code> ),
<code>is_upper_camel_case_string()</code> (in module <code>abjad.tools.stringtools.is_upper_camel_case_string</code> ),	<a href="#">292</a>
<a href="#">1126</a>	<code>iterate_runs_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_runs_in_expr</code> ),
<code>iterate_chords_in_expr()</code> (in module <code>abjad.tools.iterationtools.iterate_chords_in_expr</code> ),	<a href="#">292</a>

`iterate_scores_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_scores_in_expr`), 293  
`iterate_tuplets_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_tuplets_in_expr`), 297  
`iterate_semantic_voices_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_semantic_voices_in_expr`), 293  
`iterate_vertical_moments_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_vertical_moments_in_expr`), 297  
`iterate_sequence_cyclically()` (in module `abjad.tools.sequencetools`, `iterate_sequence_cyclically`), 906  
`iterate_voices_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_voices_in_expr`), 298  
`iterate_sequence_cyclically_from_start_to_stop()` (in module `abjad.tools.sequencetools`, `iterate_sequence_cyclically_from_start_to_stop`), 907  
`JobHandler` (class in `abjad.tools.quantizationtools`, `JobHandler.JobHandler`), 625  
`iterate_sequence_forward_and_backward_nonoverlapping()` (in module `abjad.tools.sequencetools`, `iterate_sequence_forward_and_backward_nonoverlapping`), 907  
`join_subsequences()` (in module `abjad.tools.sequencetools`, `join_subsequences`), 909  
`iterate_sequence_forward_and_backward_overlapping()` (in module `abjad.tools.sequencetools`, `iterate_sequence_forward_and_backward_overlapping`), 907  
`join_subsequences_by_sign_of_subsequence_elements()` (in module `abjad.tools.sequencetools`, `join_subsequences_by_sign_of_subsequence_elements`), 909  
`iterate_sequence_nwise_cyclic()` (in module `abjad.tools.sequencetools`, `iterate_sequence_nwise_cyclic`), 907  
`KeySignatureMark` (class in `abjad.tools.contexttools`, `KeySignatureMark.KeySignatureMark`), 58  
`iterate_sequence_nwise_strict()` (in module `abjad.tools.sequencetools`, `iterate_sequence_nwise_strict`), 908  
`label_leaves_in_expr_with_leaf_depth()` (in module `abjad.tools.labeltools`, `label_leaves_in_expr_with_leaf_depth`), 302  
`iterate_sequence_nwise_wrapped()` (in module `abjad.tools.sequencetools`, `iterate_sequence_nwise_wrapped`), 908  
`label_leaves_in_expr_with_leaf_duration()` (in module `abjad.tools.labeltools`, `label_leaves_in_expr_with_leaf_duration`), 303  
`iterate_sequence_pairwise_cyclic()` (in module `abjad.tools.sequencetools`, `iterate_sequence_pairwise_cyclic`), 908  
`label_leaves_in_expr_with_leaf_durations()` (in module `abjad.tools.labeltools`, `label_leaves_in_expr_with_leaf_durations`), 303  
`iterate_sequence_pairwise_strict()` (in module `abjad.tools.sequencetools`, `iterate_sequence_pairwise_strict`), 909  
`label_leaves_in_expr_with_leaf_indices()` (in module `abjad.tools.labeltools`, `label_leaves_in_expr_with_leaf_indices`), 304  
`iterate_sequence_pairwise_wrapped()` (in module `abjad.tools.sequencetools`, `iterate_sequence_pairwise_wrapped`), 909  
`label_leaves_in_expr_with_leaf_numbers()` (in module `abjad.tools.labeltools`, `label_leaves_in_expr_with_leaf_numbers`), 304  
`iterate_skips_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_skips_in_expr`), 294  
`label_leaves_in_expr_with_named_interval_classes()` (in module `abjad.tools.labeltools`, `label_leaves_in_expr_with_named_interval_classes`), 305  
`iterate_staves_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_staves_in_expr`), 294  
`label_leaves_in_expr_with_named_intervals()` (in module `abjad.tools.labeltools`, `label_leaves_in_expr_with_named_intervals`), 305  
`iterate_tie_chains_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_tie_chains_in_expr`), 294  
`label_leaves_in_expr_with_numbered_interval_classes()` (in module `abjad.tools.labeltools`, `label_leaves_in_expr_with_numbered_interval_classes`), 305  
`iterate_timeline_from_component()` (in module `abjad.tools.iterationtools`, `iterate_timeline_from_component`), 295  
`iterate_timeline_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_timeline_in_expr`), 295  
`iterate_topmost_tie_chains_and_components_in_expr()` (in module `abjad.tools.iterationtools`, `iterate_topmost_tie_chains_and_components_in_expr`), 296

[label\\_leaves\\_in\\_expr\\_with\\_numbered\\_intervals\(\)](#)  
 (in module abjad.tools.labeltools 311)  
[label\\_leaves\\_in\\_expr\\_with\\_numbered\\_intervals\\_in\\_expr\\_with\\_pitch\\_numbers\(\)](#)  
 (in module abjad.tools.labeltools 306)  
[label\\_leaves\\_in\\_expr\\_with\\_numbered\\_inversion\\_equivalent\\_intervals\\_in\\_expr\\_with\\_pitch\\_numbers\(\)](#)  
 (in module abjad.tools.labeltools 312)  
[label\\_leaves\\_in\\_expr\\_with\\_numbered\\_inversion\\_equivalent\\_intervals\\_in\\_expr\\_with\\_pitch\\_numbers\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 306)  
[label\\_leaves\\_in\\_expr\\_with\\_pitch\\_class\\_numbers\(\)](#)  
 (in module abjad.tools.labeltools 1396)  
[label\\_leaves\\_in\\_expr\\_with\\_pitch\\_class\\_numbers\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 306)  
[label\\_leaves\\_in\\_expr\\_with\\_pitch\\_numbers\(\)](#)  
 (in module abjad.tools.labeltools 347)  
[label\\_leaves\\_in\\_expr\\_with\\_pitch\\_numbers\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 307)  
[label\\_leaves\\_in\\_expr\\_with\\_tuplet\\_depth\(\)](#)  
 (in module abjad.tools.mathtools 428)  
[label\\_leaves\\_in\\_expr\\_with\\_tuplet\\_depth\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 307)  
[label\\_leaves\\_in\\_expr\\_with\\_written\\_leaf\\_duration\(\)](#)  
 (in module abjad.tools.mathtools 429)  
[label\\_leaves\\_in\\_expr\\_with\\_written\\_leaf\\_duration\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 308)  
[label\\_notes\\_in\\_expr\\_with\\_note\\_indices\(\)](#)  
 (in module abjad.tools.marktools 377)  
[label\\_notes\\_in\\_expr\\_with\\_note\\_indices\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 308)  
[label\\_tie\\_chains\\_in\\_expr\\_with\\_tie\\_chain\\_duration\(\)](#)  
 (in module abjad.tools.lilypondproxymarktools.LilyPondComponentPlugIn 1742)  
[label\\_tie\\_chains\\_in\\_expr\\_with\\_tie\\_chain\\_durations\(\)](#)  
 (in module abjad.tools.lilypondproxymarktools.LilyPondComponentPlugIn 1742)  
[label\\_tie\\_chains\\_in\\_expr\\_with\\_tie\\_chain\\_durations\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 309)  
[label\\_tie\\_chains\\_in\\_expr\\_with\\_written\\_tie\\_chain\\_duration\(\)](#)  
 (in module abjad.tools.lilypondproxymarktools.LilyPondComponentPlugIn 1743)  
[label\\_tie\\_chains\\_in\\_expr\\_with\\_written\\_tie\\_chain\\_durations\(\)](#)  
 (in module abjad.tools.lilypondproxymarktools.LilyPondComponentPlugIn 1743)  
[label\\_tie\\_chains\\_in\\_expr\\_with\\_written\\_tie\\_chain\\_durations\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 309)  
[label\\_vertical\\_moments\\_in\\_expr\\_with\\_interval\\_class\\_vectors\(\)](#)  
 (in module abjad.tools.lilypondparsertools.LilyPondDuration.LilyPondDuration 1697)  
[label\\_vertical\\_moments\\_in\\_expr\\_with\\_named\\_intervals\(\)](#)  
 (in module abjad.tools.lilypondparsertools.LilyPondEvent.LilyPondEvent 1697)  
[label\\_vertical\\_moments\\_in\\_expr\\_with\\_named\\_intervals\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 310)  
[label\\_vertical\\_moments\\_in\\_expr\\_with\\_numbered\\_interval\\_classes\(\)](#)  
 (in module abjad.tools.lilypondparsertools.LilyPondFraction.LilyPondFraction 1698)  
[label\\_vertical\\_moments\\_in\\_expr\\_with\\_numbered\\_interval\\_classes\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 310)  
[label\\_vertical\\_moments\\_in\\_expr\\_with\\_numbered\\_intervals\(\)](#)  
 (in module abjad.tools.lilypondparsertools.LilyPondGrammarGenerator.LilyPondGrammarGenerator 1699)  
[label\\_vertical\\_moments\\_in\\_expr\\_with\\_numbered\\_intervals\\_in\\_LaTeXOutputFormat\(\)](#)  
 (in module abjad.tools.abjadbooktools.LaTeXOutputFormat.LaTeXOutputFormat 311)  
[label\\_vertical\\_moments\\_in\\_expr\\_with\\_numbered\\_pitch\\_classes\(\)](#)  
 (in module abjad.tools.lilypondproxymarktools.LilyPondGrobOverrideComponentPlugIn 1744)

LilyPondGrobProxy (class in abjad.tools.lilypondproxytools.LilyPondGrobProxy), 276  
1745 log\_render\_lilypond\_input() (in module abjad.tools.iotools.log\_render\_lilypond\_input),  
LilyPondGrobProxyContextWrapper (class in abjad.tools.lilypondproxytools.LilyPondGrobProxyContextWrapper), 1746  
1746 ly() (in module abjad.tools.iotools.ly), 277  
LilyPondLanguageToken (class in abjad.tools.lilypondfiletools.LilyPondLanguageToken), 354  
354 make\_accelerating\_notes\_with\_lilypond\_multipliers() (in module abjad.tools.lilypondparsertools.LilyPondLexicalDefinition), 1700  
1700 make\_basic\_lilypond\_file() (in module abjad.tools.lilypondfiletools.make\_basic\_lilypond\_file), 1747  
1747 make\_big\_centered\_page\_number\_markup() (in module abjad.tools.markuptools.make\_big\_centered\_page\_number\_markup), 1703  
1703 make\_blank\_line\_markup() (in module abjad.tools.markuptools.make\_blank\_line\_markup), 1661  
1661 make\_centered\_title\_markup() (in module abjad.tools.markuptools.make\_centered\_title\_markup), 1712  
1712 make\_desordre\_cell() (in module abjad.demos.desordre.make\_desordre\_cell), 1748  
1748 make\_desordre\_lilypond\_file() (in module abjad.demos.desordre.make\_desordre\_lilypond\_file), 1371  
1371 make\_desordre\_measure() (in module abjad.demos.desordre.make\_desordre\_measure), 1371  
1371 make\_desordre\_pitches() (in module abjad.demos.desordre.make\_desordre\_pitches), 1371  
1371 make\_desordre\_score() (in module abjad.demos.desordre.make\_desordre\_score), 1371  
1371 make\_desordre\_staff() (in module abjad.demos.desordre.make\_desordre\_staff), 1371  
1371 make\_dynamic\_spanner\_below\_with\_nib\_at\_right() (in module abjad.tools.spannertools.make\_dynamic\_spanner\_below\_with\_nib\_at\_right), 614  
614 make\_empty\_piano\_score() (in module abjad.tools.scoretools.make\_empty\_piano\_score), 863  
863 make\_floating\_time\_signature\_lilypond\_file() (in module abjad.tools.lilypondfiletools.make\_floating\_time\_signature\_lilypond\_file), 615  
615 make\_gridded\_test\_rhythm() (in module abjad.tools.timesignaturetools.make\_gridded\_test\_rhythm), 1233  
1233

make_leaves() (in module abjad.tools.leaftools.make_leaves), 323	(in module abjad.tools.scoretools.make_piano_sketch_score_from_leaves), 864
make_leaves_from_talea() (in module abjad.tools.leaftools.make_leaves_from_talea), 326	make_quarter_notes_with_lilypond_duration_multiplier() (in module abjad.tools.notetools.make_quarter_notes_with_lilypond_duration_multiplier), 481
make_ligeti_example_lilypond_file() (in module abjad.tools.documentationtools.make_ligeti_example_lilypond_file), 1651	make_reference_manual_graphviz_graph() (in module abjad.tools.documentationtools.make_reference_manual_graphviz_graph), 1651
make_lilypond_file() (in module abjad.demos.ferneyhough.make_lilypond_file), 1373	make_reference_manual_lilypond_file() (in module abjad.tools.documentationtools.make_reference_manual_lilypond_file), 1651
make_lilypond_override_string() (in module abjad.tools.formattools.make_lilypond_override_string), 1685	make_repeated_notes() (in module abjad.tools.notetools.make_repeated_notes), 482
make_lilypond_revert_string() (in module abjad.tools.formattools.make_lilypond_revert_string), 1685	make_repeated_notes_from_time_signature() (in module abjad.tools.notetools.make_repeated_notes_from_time_signature), 482
make_measures_with_full_measure_spacer_skips() (in module abjad.tools.measuretools.make_measures_with_full_measure_spacer_skips), 449	make_repeated_notes_from_time_signatures() (in module abjad.tools.notetools.make_repeated_notes_from_time_signatures), 482
make_mozart_lilypond_file() (in module abjad.demos.mozart.make_mozart_lilypond_file), 1375	make_repeated_notes_with_shorter_notes_at_end() (in module abjad.tools.notetools.make_repeated_notes_with_shorter_notes_at_end), 482
make_mozart_measure() (in module abjad.demos.mozart.make_mozart_measure), 1375	make_repeated_rests_from_time_signatures() (in module abjad.tools.resttools.make_repeated_rests_from_time_signatures), 710
make_mozart_measure_corpus() (in module abjad.demos.mozart.make_mozart_measure_corpus), 1375	make_repeated_skips_from_time_signatures() (in module abjad.tools.skiptools.make_repeated_skips_from_time_signatures), 941
make_mozart_score() (in module abjad.demos.mozart.make_mozart_score), 1375	make_rests() (in module abjad.tools.resttools.make_rests), 710
make_multimeasure_rests() (in module abjad.tools.resttools.make_multimeasure_rests), 710	make_rhythmic_sketch_staff() (in module abjad.tools.stafftools.make_rhythmic_sketch_staff), 1120
make_n_middle_c_centered_pitches() (in module abjad.tools.pitchtools.make_n_middle_c_centered_pitches), 616	make_row_of_nested_tuplets() (in module abjad.demos.ferneyhough.make_row_of_nested_tuplets), 1373
make_nested_tuplet() (in module abjad.demos.ferneyhough.make_nested_tuplet), 1373	make_rows_of_nested_tuplets() (in module abjad.demos.ferneyhough.make_rows_of_nested_tuplets), 1373
make_notes() (in module abjad.tools.notetools.make_notes), 480	make_score() (in module abjad.demos.ferneyhough.make_score), 1373
make_notes_with_multiplied_durations() (in module abjad.tools.notetools.make_notes_with_multiplied_durations), 480	make_skips_with_multiplied_durations() (in module abjad.tools.skiptools.make_skips_with_multiplied_durations), 941
make_part_lilypond_file() (in module abjad.demos.part.make_part_lilypond_file), 1379	make_solid_text_spanner_with_nib() (in module abjad.tools.spannertools.make_solid_text_spanner_with_nib), 1105
make_percussion_note() (in module abjad.tools.notetools.make_percussion_note), 481	make_spacing_vector() (in module abjad.tools.spacingtools.make_spacing_vector), 1105
make_piano_score_from_leaves() (in module abjad.tools.scoretools.make_piano_score_from_leaves), 863	
make_piano_sketch_score_from_leaves() (in module abjad.tools.scoretools.make_piano_sketch_score_from_leaves), 864	



jad.tools.layouttools.make_spacing_vector),	437
316	MeasuredComplexBeamSpanner (class in ab-
make_test_intervals() (in module ab-	jad.tools.spannertools.MeasuredComplexBeamSpanner.MeasuredComplexBeamSpanner),
jad.tools.timeintervaltools.make_test_intervals),	1036
1183	MeasurewiseAttackPointOptimizer (class in ab-
make_test_time_segments() (in module ab-	jad.tools.quantizationtools.MeasurewiseAttackPointOptimizer.MeasurewiseAttackPointOptimizer),
jad.tools.quantizationtools.make_test_time_segments),	644
704	MeasurewiseQSchema (class in ab-
make_text_alignment_example_lilypond_file()	jad.tools.quantizationtools.MeasurewiseQSchema.MeasurewiseQSchema),
(in module ab-	645
jad.tools.documentationtools.make_text_alignment_example_lilypond_file),	(class in ab-
1651	jad.tools.quantizationtools.MeasurewiseQSchemaItem.MeasurewiseQSchemaItem),
make_tied_leaf() (in module ab-	649
jad.tools.leaftools.make_tied_leaf),	327
make_time_signature_context_block() (in module ab-	MeasurewiseQTarget (class in ab-
jad.tools.lilypondfiletools.make_time_signature_context_block),	jad.tools.quantizationtools.MeasurewiseQTarget.MeasurewiseQTarget),
364	merge_duration_sequences() (in module ab-
make_vertically_adjusted_composer_markup()	jad.tools.sequencetools.merge_duration_sequences),
(in module ab-	910
jad.tools.markuptools.make_vertically_adjusted_composer_markup),	(class in ab-
398	jad.tools.timesignaturetools.MetricalHierarchy.MetricalHierarchy),
MakeNewClassTemplateScript (class in ab-	1215
jad.tools.developerscripttools.MakeNewClassTemplateScript),	MetricalHierarchyNewClassTemplateScript (class in ab-
1497	jad.tools.timesignaturetools.MetricalHierarchyInventory.MetricalHierarchyInventory),
MakeNewFunctionTemplateScript (class in ab-	1221
jad.tools.developerscripttools.MakeNewFunctionTemplateScript),	MetricalKernel (class in ab-
1499	jad.tools.timesignaturetools.MetricalKernel.MetricalKernel),
Maker (class in abjad.tools.abctools.Maker.Maker),	1225
1384	MezzoSopranoVoice (class in ab-
map_sequence_elements_to_canonic_tuples()	jad.tools.instrumenttools.MezzoSopranoVoice.MezzoSopranoVoice),
(in module ab-	215
jad.tools.sequencetools.map_sequence_elements_to_canonic_tuples),	(class in ab-
909	jad.tools.lilypondfiletools.MIDIBlock.MIDIBlock),
map_sequence_elements_to_numbered_sublists()	356
(in module ab-	MisduratedMeasureCheck (class in ab-
jad.tools.sequencetools.map_sequence_elements_to_numbered_sublists),	jad.tools.wellformednesstools.MisduratedMeasureCheck.MisduratedMeasureCheck),
910	1358
Marimba (class in ab-	MisfilledMeasureCheck (class in ab-
jad.tools.instrumenttools.Marimba.Marimba),	jad.tools.wellformednesstools.MisfilledMeasureCheck.MisfilledMeasureCheck),
212	1359
Mark (class in abjad.tools.marktools.Mark.Mark),	382
Markup (class in ab-	MispitchedTieCheck (class in ab-
jad.tools.markuptools.Markup.Markup),	jad.tools.wellformednesstools.MispitchedTieCheck.MispitchedTieCheck),
387	1360
MarkupCommand (class in ab-	MisrepresentedFlagCheck (class in ab-
jad.tools.markuptools.MarkupCommand.MarkupCommand),	jad.tools.wellformednesstools.MisrepresentedFlagCheck.MisrepresentedFlagCheck),
390	61
MarkupInventory (class in ab-	MissingInstrumentError (class in ab-
jad.tools.markuptools.MarkupInventory.MarkupInventory),	jad.tools.exceptiontools.MissingInstrumentError),
392	662
mask_intervals_with_intervals() (in module ab-	MissingMarkError (class in ab-
jad.tools.timeintervaltools.mask_intervals_with_intervals),	jad.tools.exceptiontools.MissingMarkError),
1183	663
Matrix (class in ab-	MissingMeasureError (class in ab-
jad.tools.datastructuretools.Matrix.Matrix),	jad.tools.exceptiontools.MissingMeasureError),
1436	1664
Measure (class in ab-	MissingNamedComponentError (class in ab-
jad.tools.measuretools.Measure.Measure),	jad.tools.exceptiontools.MissingNamedComponentError),
	1665

MissingNoteHeadError	(class in abjad.tools.exceptiontools.MissingNoteHeadError), 1666	NamedIntervalClass	(class in abjad.tools.pitchtools.NamedIntervalClass.NamedIntervalClass), 539
MissingParentCheck	(class in abjad.tools.wellformednesstools.MissingParentCheck.MissingParentCheck), 1362	NamedInversionEquivalentIntervalClass	(class in abjad.tools.pitchtools.NamedInversionEquivalentIntervalClass.NamedInversionEquivalentIntervalClass), 541
MissingPitchError	(class in abjad.tools.exceptiontools.MissingPitchError), 1667	NamedPitch	(class in abjad.tools.pitchtools.NamedPitch.NamedPitch), 542
MissingSpannerError	(class in abjad.tools.exceptiontools.MissingSpannerError), 1668	NamedPitchClass	(class in abjad.tools.pitchtools.NamedPitchClass.NamedPitchClass), 547
MissingTempoError	(class in abjad.tools.exceptiontools.MissingTempoError), 1669	NaturalHarmonic	(class in abjad.tools.notetools.NaturalHarmonic.NaturalHarmonic), 470
Mode	(class in abjad.tools.tonalanalysistools.Mode.Mode), 1294	negate_absolute_value_of_sequence_elements_at_indices()	(in module abjad.tools.sequencetools.negate_absolute_value_of_sequence_elements_at_indices()), 910
ModuleCrawler	(class in abjad.tools.documentationtools.ModuleCrawler.ModuleCrawler), 1566	negate_absolute_value_of_sequence_elements_cyclically()	(in module abjad.tools.sequencetools.negate_absolute_value_of_sequence_elements_cyclically()), 910
move_full_measure_tuplet_prolation_to_measure_time_signature()	(in module abjad.tools.measuretools.move_full_measure_tuplet_prolation_to_measure_time_signature()), 449	negate_absolute_value_of_sequence_elements_time_indicators()	(in module abjad.tools.sequencetools.negate_absolute_value_of_sequence_elements_time_indicators()), 911
move_measure_prolation_to_full_measure_tuplet()	(in module abjad.tools.measuretools.move_measure_prolation_to_full_measure_tuplet()), 450	negate_sequence_elements_cyclically()	(in module abjad.tools.sequencetools.negate_sequence_elements_cyclically()), 911
MultimeasureRest	(class in abjad.tools.resttools.MultimeasureRest.MultimeasureRest), 705	NegativeInfinity	(class in abjad.tools.mathtools.NegativeInfinity.NegativeInfinity), 404
MultipartBeamSpanner	(class in abjad.tools.spannertools.MultipartBeamSpanner.MultipartBeamSpanner), 1044	NestedMeasureCheck	(class in abjad.tools.wellformednesstools.NestedMeasureCheck.NestedMeasureCheck), 1363
Multiplier	(class in abjad.tools.durationtools.Multiplier.Multiplier), 87	next_integer_partition()	(in module abjad.tools.mathtools.next_integer_partition()), 430
Music	(class in abjad.tools.lilypondparsertools.Music.Music), 1691	NonattributedBlock	(class in abjad.tools.lilypondfiletools.NonattributedBlock.NonattributedBlock), 332
MusicGlyph	(class in abjad.tools.markuptools.MusicGlyph.MusicGlyph), 396	NonreducedFraction	(class in abjad.tools.mathtools.NonreducedFraction.NonreducedFraction), 406
mutate()	(in module abjad.tools.mutationtools.mutate), 467	NonreducedRatio	(class in abjad.tools.mathtools.NonreducedRatio.NonreducedRatio), 413
<b>N</b>		Note	(class in abjad.tools.notetools.Note.Note), 474
NaiveAttackPointOptimizer	(class in abjad.tools.quantizationtools.NaiveAttackPointOptimizer.NaiveAttackPointOptimizer), 652	NoteHead	(class in abjad.tools.notetools.NoteHead.NoteHead), 477
named_pitch_and_clef_to_staff_position_number()	(in module abjad.tools.pitchtools.named_pitch_and_clef_to_staff_position_number), 616	NoteRhythmMaker	(class in abjad.tools.rhythm makertools.NoteRhythmMaker.NoteRhythmMaker), 745
NamedInterval	(class in abjad.tools.pitchtools.NamedInterval.NamedInterval), 537	notes_and_chords_are_in_range()	(in module abjad.tools.instrumenttools.notes_and_chords_are_in_range), 270
		notes_and_chords_are_on_expected_clefs()	



(in module abjad.tools.instrumenttools.notes\_and\_chords\_are\_offset\_expanded\_before\_timespan\_stops() 1203  
 jad.tools.instrumenttools.notes\_and\_chords\_are\_offset\_expanded\_before\_timespan\_stops() 270  
 (in module abjad.tools.timerelationtools.offset\_happens\_before\_timespan\_stops() 1203  
 NullAttackPointOptimizer (class in abjad.tools.quantizationtools.NullAttackPointOptimizer.NullAttackPointOptimizer), 653  
 jad.tools.quantizationtools.NullAttackPointOptimizer.NullAttackPointOptimizer, 653  
 offset\_happens\_during\_timespan() (in module abjad.tools.timerelationtools.offset\_happens\_during\_timespan), 1204  
 numbered\_inversion\_equivalent\_interval\_class\_dictionary() (in module abjad.tools.pitchtools.numbered\_inversion\_equivalent\_interval\_class\_dictionary), 616  
 jad.tools.pitchtools.numbered\_inversion\_equivalent\_interval\_class\_dictionary, 616  
 (in module abjad.tools.timerelationtools.offset\_happens\_when\_timespan\_starts() (in module abjad.tools.timerelationtools.offset\_happens\_when\_timespan\_starts), 1204  
 NumberedInterval (class in abjad.tools.pitchtools.NumberedInterval.NumberedInterval), 551  
 jad.tools.pitchtools.NumberedInterval.NumberedInterval, 551  
 NumberedIntervalClass (class in abjad.tools.pitchtools.NumberedIntervalClass.NumberedIntervalClass), 553  
 jad.tools.pitchtools.NumberedIntervalClass.NumberedIntervalClass, 553  
 OffsetIntervalClass (class in abjad.tools.timerelationtools.OffsetIntervalClass.OffsetIntervalClass), 1192  
 OffsetTimespanTimeRelation (class in abjad.tools.timerelationtools.OffsetTimespanTimeRelation.OffsetTimespanTimeRelation), 1192  
 NumberedInversionEquivalentIntervalClass (class in abjad.tools.pitchtools.NumberedInversionEquivalentIntervalClass.NumberedInversionEquivalentIntervalClass), 555  
 jad.tools.pitchtools.NumberedInversionEquivalentIntervalClass.NumberedInversionEquivalentIntervalClass, 555  
 OmissionIndicator (class in abjad.tools.pitchtools.OmissionIndicator.OmissionIndicator), 1295  
 NumberedPitch (class in abjad.tools.pitchtools.NumberedPitch.NumberedPitch), 556  
 jad.tools.pitchtools.NumberedPitch.NumberedPitch, 556  
 OrdinalConstant (class in abjad.tools.datastructuretools.OrdinalConstant.OrdinalConstant), 561  
 jad.tools.datastructuretools.OrdinalConstant.OrdinalConstant, 561  
 OutputBurnishedTaleaRhythmMaker (class in abjad.tools.rhythmtools.OutputBurnishedTaleaRhythmMaker.OutputBurnishedTaleaRhythmMaker), 1389  
 jad.tools.rhythmtools.OutputBurnishedTaleaRhythmMaker.OutputBurnishedTaleaRhythmMaker, 1389  
 OutputFormat (class in abjad.tools.abjadbooktools.OutputFormat.OutputFormat), 1389  
 jad.tools.abjadbooktools.OutputFormat.OutputFormat, 1389  
 O (1389)  
 Oboe (class in abjad.tools.instrumenttools.Oboe.Oboe), 218  
 jad.tools.instrumenttools.Oboe.Oboe, 218  
 OctavationSpanner (class in abjad.tools.spannertools.OctavationSpanner.OctavationSpanner), 1050  
 jad.tools.spannertools.OctavationSpanner.OctavationSpanner, 1050  
 Octave (class in abjad.tools.pitchtools.Octave.Octave), 566  
 jad.tools.pitchtools.Octave.Octave, 566  
 OctaveTranspositionMapping (class in abjad.tools.pitchtools.OctaveTranspositionMapping.OctaveTranspositionMapping), 569  
 jad.tools.pitchtools.OctaveTranspositionMapping.OctaveTranspositionMapping, 569  
 OctaveTranspositionMappingComponent (class in abjad.tools.pitchtools.OctaveTranspositionMappingComponent.OctaveTranspositionMappingComponent), 573  
 jad.tools.pitchtools.OctaveTranspositionMappingComponent.OctaveTranspositionMappingComponent, 573  
 OctaveTranspositionMappingInventory (class in abjad.tools.pitchtools.OctaveTranspositionMappingInventory.OctaveTranspositionMappingInventory), 575  
 jad.tools.pitchtools.OctaveTranspositionMappingInventory.OctaveTranspositionMappingInventory, 575  
 Offset (class in abjad.tools.durationtools.Offset.Offset), 96  
 jad.tools.durationtools.Offset.Offset, 96  
 offset\_happens\_after\_timespan\_starts() (in module abjad.tools.timerelationtools.offset\_happens\_after\_timespan\_starts), 1202  
 jad.tools.timerelationtools.offset\_happens\_after\_timespan\_starts, 1202  
 offset\_happens\_after\_timespan\_stops() (in module abjad.tools.timerelationtools.offset\_happens\_after\_timespan\_stops), 1202  
 jad.tools.timerelationtools.offset\_happens\_after\_timespan\_stops, 1202  
 offset\_happens\_before\_timespan\_starts() (in module abjad.tools.timerelationtools.offset\_happens\_before\_timespan\_starts), 1202  
 jad.tools.timerelationtools.offset\_happens\_before\_timespan\_starts, 1202  
 (in module abjad.tools.timerelationtools.offset\_happens\_before\_timespan\_starts), 1202  
 jad.tools.timerelationtools.offset\_happens\_before\_timespan\_starts, 1202  
 overwrite\_sequence\_elements\_at\_indices() (in module abjad.tools.sequencetools.overwrite\_sequence\_elements\_at\_indices), 911  
 jad.tools.sequencetools.overwrite\_sequence\_elements\_at\_indices, 911

## P

- `p()` (in module `abjad.tools.iotools.p`), 277
- `pair_duration_sequence_elements_with_input_pair_values()` (in module `abjad.tools.sequencetools.pair_duration_sequence_elements_with_input_pair_values`), 911
- `PaperBlock` (class in `abjad.tools.lilypondfiletools.PaperBlock.PaperBlock`), 359
- `ParallelJobHandler` (class in `abjad.tools.quantizationtools.ParallelJobHandler.ParallelJobHandler`), 654
- `ParallelJobHandlerWorker` (class in `abjad.tools.quantizationtools.ParallelJobHandlerWorker.ParallelJobHandlerWorker`), 655
- `Parentage` (class in `abjad.tools.selectiontools.Parentage.Parentage`), 871
- `parse_reduced_ly_syntax()` (in module `abjad.tools.lilypondparsertools.parse_reduced_ly_syntax`), 1740
- `parse_rtm_syntax()` (in module `abjad.tools.rhythmtreetools.parse_rtm_syntax`), 799
- `Parser` (class in `abjad.tools.abctools.Parser.Parser`), 1385
- `PartCantusScoreTemplate` (class in `abjad.demos.part.PartCantusScoreTemplate.PartCantusScoreTemplate`), 1377
- `partition_integer_by_ratio()` (in module `abjad.tools.mathtools.partition_integer_by_ratio`), 430
- `partition_integer_into_canonic_parts()` (in module `abjad.tools.mathtools.partition_integer_into_canonic_parts`), 430
- `partition_integer_into_halves()` (in module `abjad.tools.mathtools.partition_integer_into_halves`), 431
- `partition_integer_into_parts_less_than_double()` (in module `abjad.tools.mathtools.partition_integer_into_parts_less_than_double`), 432
- `partition_integer_into_units()` (in module `abjad.tools.mathtools.partition_integer_into_units`), 432
- `partition_sequence_by_backgrounded_weights()` (in module `abjad.tools.sequencetools.partition_sequence_by_backgrounded_weights`), 912
- `partition_sequence_by_counts()` (in module `abjad.tools.sequencetools.partition_sequence_by_counts`), 912
- `partition_sequence_by_ratio_of_lengths()` (in module `abjad.tools.sequencetools.partition_sequence_by_ratio_of_lengths`), 914
- `partition_sequence_by_ratio_of_weights()` (in module `abjad.tools.sequencetools.partition_sequence_by_ratio_of_weights`), 914
- `partition_sequence_by_restricted_growth_function()` (in module `abjad.tools.sequencetools.partition_sequence_by_restricted_growth_function`), 915
- `partition_sequence_by_sign_of_elements()` (in module `abjad.tools.sequencetools.partition_sequence_by_sign_of_elements`), 915
- `partition_sequence_by_value_of_elements()` (in module `abjad.tools.sequencetools.partition_sequence_by_value_of_elements`), 916
- `partition_sequence_by_weights_at_least()` (in module `abjad.tools.sequencetools.partition_sequence_by_weights_at_least`), 916
- `partition_sequence_by_weights_at_most()` (in module `abjad.tools.sequencetools.partition_sequence_by_weights_at_most`), 916
- `partition_sequence_by_weights_exactly()` (in module `abjad.tools.sequencetools.partition_sequence_by_weights_exactly`), 917
- `partition_sequence_extended_to_counts()` (in module `abjad.tools.sequencetools.partition_sequence_extended_to_counts`), 918
- `PartitionError` (class in `abjad.tools.exceptiontools.PartitionError`), 1671
- `PayloadTree` (class in `abjad.tools.datastructuretools.PayloadTree.PayloadTree`), 1439
- `pdf()` (in module `abjad.tools.iotools.pdf`), 277
- `Performer` (class in `abjad.tools.scoretools.Performer.Performer`), 832
- `PerformerInventory` (class in `abjad.tools.scoretools.PerformerInventory.PerformerInventory`), 838
- `permute_named_pitch_carrier_list_by_twelve_tone_row()` (in module `abjad.tools.pitchtools.permute_named_pitch_carrier_list_by_twelve_tone_row`), 617
- `permute_sequence()` (in module `abjad.tools.sequencetools.permute_sequence`), 918
- `PhrasingSlurSpanner` (class in `abjad.tools.spannertools.PhrasingSlurSpanner.PhrasingSlurSpanner`), 1057
- `Piano` (class in `abjad.tools.instrumenttools.Piano.Piano`), 221
- `PianoPedalSpanner` (class in `abjad.tools.spannertools.PianoPedalSpanner.PianoPedalSpanner`), 1057

1063		jad.tools.stringtools.pluralize_string), 1127
PianoStaff	(class in abjad.tools.scoretools.PianoStaff.PianoStaff), 842	profile_expr() (in module abjad.tools.iotools.profile_expr), 278
Piccolo	(class in abjad.tools.instrumenttools.Piccolo.Piccolo), 224	PyTestScript (class in abjad.tools.developerscripttools.PyTestScript.PyTestScript), 1501
Pipe	(class in abjad.tools.documentationtools.Pipe.Pipe), 1567	<b>Q</b>
Pitch	(class in abjad.tools.pitchtools.Pitch.Pitch), 504	QEvent (class in abjad.tools.quantizationtools.QEvent.QEvent), 626
PitchArray	(class in abjad.tools.pitcharraytools.PitchArray.PitchArray), 485	QEventProxy (class in abjad.tools.quantizationtools.QEventProxy.QEventProxy), 658
PitchArrayCell	(class in abjad.tools.pitcharraytools.PitchArrayCell.PitchArrayCell), 489	QEventSequence (class in abjad.tools.quantizationtools.QEventSequence.QEventSequence), 659
PitchArrayColumn	(class in abjad.tools.pitcharraytools.PitchArrayColumn.PitchArrayColumn), 491	QGrid (class in abjad.tools.quantizationtools.QGrid.QGrid), 665
PitchArrayInventory	(class in abjad.tools.pitcharraytools.PitchArrayInventory.PitchArrayInventory), 493	QGridContainer (class in abjad.tools.quantizationtools.QGridContainer.QGridContainer), 668
PitchArrayRow	(class in abjad.tools.pitcharraytools.PitchArrayRow.PitchArrayRow), 497	QGridLeaf (class in abjad.tools.quantizationtools.QGridLeaf.QGridLeaf), 680
PitchClass	(class in abjad.tools.pitchtools.PitchClass.PitchClass), 507	QSchema (class in abjad.tools.quantizationtools.QSchema.QSchema), 628
PitchClassSegment	(class in abjad.tools.pitchtools.PitchClassSegment.PitchClassSegment), 579	QSchemaItem (class in abjad.tools.quantizationtools.QSchemaItem.QSchemaItem), 629
PitchClassSet	(class in abjad.tools.pitchtools.PitchClassSet.PitchClassSet), 583	QTarget (class in abjad.tools.quantizationtools.QTarget.QTarget), 631
PitchClassVector	(class in abjad.tools.pitchtools.PitchClassVector.PitchClassVector), 587	QTargetBeat (class in abjad.tools.quantizationtools.QTargetBeat.QTargetBeat), 685
PitchedQEvent	(class in abjad.tools.quantizationtools.PitchedQEvent.PitchedQEvent), 657	QTargetMeasure (class in abjad.tools.quantizationtools.QTargetMeasure.QTargetMeasure), 687
PitchRange	(class in abjad.tools.pitchtools.PitchRange.PitchRange), 589	QualityIndicator (class in abjad.tools.tonalanalysisistools.QualityIndicator.QualityIndicator), 1296
PitchRangeInventory	(class in abjad.tools.pitchtools.PitchRangeInventory.PitchRangeInventory), 592	QuantizationJob (class in abjad.tools.quantizationtools.QuantizationJob.QuantizationJob), 690
PitchSegment	(class in abjad.tools.pitchtools.PitchSegment.PitchSegment), 596	Quantizer (class in abjad.tools.quantizationtools.Quantizer.Quantizer), 693
PitchSet	(class in abjad.tools.pitchtools.PitchSet.PitchSet), 600	<b>R</b>
PitchVector	(class in abjad.tools.pitchtools.PitchVector.PitchVector), 603	Ratio (class in abjad.tools.mathtools.Ratio.Ratio), 415
play() (in module abjad.tools.iotools.play), 278		read_test_output() (in module abjad.tools.testtools.read_test_output), 1753
plot() (in module abjad.tools.iotools.plot), 278		RedirectedStreams (class in abjad.tools.iotools.RedirectedStreams.RedirectedStreams), 273
pluralize_string() (in module ab-		

redo() (in module abjad.tools.iotools.redo), 279

ReducedLyParser (class in abjad.tools.lilypondparsertools.ReducedLyParser.ReducedLyParser), 1729

replace\_contents\_of\_measures\_in\_expr() (in module abjad.tools.measuretools.replace\_contents\_of\_measures\_in\_expr), 450

register\_pitch\_class\_numbers\_by\_pitch\_number\_aggregate() (in module abjad.tools.pitchtools.register\_pitch\_class\_numbers\_by\_pitch\_number\_aggregate), 617

replace\_sequence\_elements\_cyclically\_with\_new\_material() (in module abjad.tools.replace\_sequence\_elements\_cyclically\_with\_new\_material), 922

remove\_markup\_from\_leaves\_in\_expr() (in module abjad.tools.labeltools.remove\_markup\_from\_leaves\_in\_expr), 312

ReplaceInFilesScript (class in abjad.tools.developerscripttools.ReplaceInFilesScript.ReplaceInFilesScript), 1505

remove\_powers\_of\_two() (in module abjad.tools.mathtools.remove\_powers\_of\_two), 433

report\_component\_format\_contributions() (in module abjad.tools.formattools.report\_component\_format\_contributions), 1685

remove\_sequence\_elements\_at\_indices() (in module abjad.tools.sequencetools.remove\_sequence\_elements\_at\_indices), 918

report\_integer\_tempo\_rewrite\_pairs() (in module abjad.tools.tempotools.report\_integer\_tempo\_rewrite\_pairs), 1131

remove\_sequence\_elements\_at\_indices\_cyclically() (in module abjad.tools.sequencetools.remove\_sequence\_elements\_at\_indices\_cyclically), 919

report\_spanner\_format\_contributions() (in module abjad.tools.formattools.report\_spanner\_format\_contributions), 1685

requires() (in module abjad.tools.decoratortools.requires), 1477

remove\_subsequence\_of\_weight\_at\_index() (in module abjad.tools.sequencetools.remove\_subsequence\_of\_weight\_at\_index), 919

ResidueClass (class in abjad.tools.sievetools.ResidueClass.ResidueClass), 934

RenameModulesScript (class in abjad.tools.developerscripttools.RenameModulesScript.RenameModulesScript), 1503

resolve\_overlaps\_between\_nonoverlapping\_trees() (in module abjad.tools.timeintervaltools.resolve\_overlaps\_between\_nonoverlapping\_trees), 1184

repeat\_runs\_in\_sequence\_to\_count() (in module abjad.tools.sequencetools.repeat\_runs\_in\_sequence\_to\_count), 919

Rest (class in abjad.tools.resttools.Rest.Rest), 708

ReSTAutodocDirective (class in abjad.tools.documentationtools.ReSTAutodocDirective.ReSTAutodocDirective), 920

repeat\_sequence\_elements\_at\_indices() (in module abjad.tools.sequencetools.repeat\_sequence\_elements\_at\_indices), 920

ReSTAutosummaryDirective (class in abjad.tools.documentationtools.ReSTAutosummaryDirective.ReSTAutosummaryDirective), 1578

repeat\_sequence\_elements\_at\_indices\_cyclically() (in module abjad.tools.sequencetools.repeat\_sequence\_elements\_at\_indices\_cyclically), 920

ReSTAutosummaryItem (class in abjad.tools.documentationtools.ReSTAutosummaryItem.ReSTAutosummaryItem), 1587

repeat\_sequence\_elements\_n\_times\_each() (in module abjad.tools.sequencetools.repeat\_sequence\_elements\_n\_times\_each), 920

ReSTDirective (class in abjad.tools.documentationtools.ReSTDirective.ReSTDirective), 1522

repeat\_sequence\_n\_times() (in module abjad.tools.sequencetools.repeat\_sequence\_n\_times), 921

ReSTDDocument (class in abjad.tools.documentationtools.ReSTDDocument.ReSTDDocument), 1590

repeat\_sequence\_to\_length() (in module abjad.tools.sequencetools.repeat\_sequence\_to\_length), 921

ReSTHeading (class in abjad.tools.documentationtools.ReSTHeading.ReSTHeading), 1599

repeat\_sequence\_to\_weight\_at\_least() (in module abjad.tools.sequencetools.repeat\_sequence\_to\_weight\_at\_least), 921

ReSTHorizontalRule (class in abjad.tools.documentationtools.ReSTHorizontalRule.ReSTHorizontalRule), 1603

repeat\_sequence\_to\_weight\_at\_most() (in module abjad.tools.sequencetools.repeat\_sequence\_to\_weight\_at\_most), 921

ReSTInheritanceDiagram (class in abjad.tools.documentationtools.ReSTInheritanceDiagram.ReSTInheritanceDiagram), 1606

repeat\_sequence\_to\_weight\_exactly() (in module abjad.tools.sequencetools.repeat\_sequence\_to\_weight\_exactly), 922

ReSTLineageDirective (class in abjad.tools.documentationtools.ReSTLineageDirective.ReSTLineageDirective), 1615

ReSTOnlyDirective	(class in abjad.tools.documentationtools.ReSTOnlyDirective.ReSTOnlyDirective), 1624	rotate_sequence()	(in module abjad.tools.sequencetools.rotate_sequence), 923
ReSTOutputFormat	(class in abjad.tools.abjadbooktools.ReSTOutputFormat.ReSTOutputFormat), 1397	run_abjad()	(in module abjad.tools.iotools.run_abjad), 460
ReSTParagraph	(class in abjad.tools.documentationtools.ReSTParagraph.ReSTParagraph), 1633	run_abjadbook()	(in module abjad.tools.developerscripttools.run_abjadbook), 451
RestRhythmMaker	(class in abjad.tools.rhythmmakertools.RestRhythmMaker.RestRhythmMaker), 760	run_abjdev()	(in module abjad.tools.developerscripttools.run_abjdev), 451
ReSTTOCDirective	(class in abjad.tools.documentationtools.ReSTTOCDirective.ReSTTOCDirective), 1637	run_lilypond()	(in module abjad.tools.iotools.run_lilypond), 279
ReSTTOCItem	(class in abjad.tools.documentationtools.ReSTTOCItem.ReSTTOCItem), 1646	RunDoctestsScript	(class in abjad.tools.developerscripttools.RunDoctestsScript.RunDoctestsScript), 1507
retain_sequence_elements_at_indices()	(in module abjad.tools.sequencetools.retain_sequence_elements_at_indices), 922	save_last_ly_as()	(in module abjad.tools.iotools.save_last_ly_as), 279
retain_sequence_elements_at_indices_cyclically()	(in module abjad.tools.sequencetools.retain_sequence_elements_at_indices_cyclically), 922	save_last_pdf_as()	(in module abjad.tools.iotools.save_last_pdf_as), 279
reverse_sequence()	(in module abjad.tools.sequencetools.reverse_sequence), 923	Scale	(class in abjad.tools.tonalanalysistools.Scale.Scale), 1298
reverse_sequence_elements()	(in module abjad.tools.sequencetools.reverse_sequence_elements), 923	scale_measure_denominator_and_adjust_measure_contents()	(in module abjad.tools.measuretools.scale_measure_denominator_and_adjust_measure_contents), 450
rewrite_duration_under_new_tempo()	(in module abjad.tools.tempotools.rewrite_duration_under_new_tempo), 1131	ScaleDegree	(class in abjad.tools.tonalanalysistools.ScaleDegree.ScaleDegree), 1302
rewrite_integer_tempo()	(in module abjad.tools.tempotools.rewrite_integer_tempo), 1132	Scheme	(class in abjad.tools.schemetools.Scheme.Scheme), 801
RhythmicStaff	(class in abjad.tools.stafftools.RhythmicStaff.RhythmicStaff), 1107	SchemeAssociativeList	(class in abjad.tools.schemetools.SchemeAssociativeList.SchemeAssociativeList), 804
RhythmMaker	(class in abjad.tools.rhythmmakertools.RhythmMaker.RhythmMaker), 723	SchemeColor	(class in abjad.tools.schemetools.SchemeColor.SchemeColor), 806
RhythmTreeContainer	(class in abjad.tools.rhythmtreetools.RhythmTreeContainer.RhythmTreeContainer), 778	SchemeMoment	(class in abjad.tools.schemetools.SchemeMoment.SchemeMoment), 808
RhythmTreeLeaf	(class in abjad.tools.rhythmtreetools.RhythmTreeLeaf.RhythmTreeLeaf), 791	SchemePair	(class in abjad.tools.schemetools.SchemePair.SchemePair), 810
RhythmTreeNode	(class in abjad.tools.rhythmtreetools.RhythmTreeNode.RhythmTreeNode), 773	SchemeParser	(class in abjad.tools.lilypondparsertools.SchemeParser.SchemeParser), 1735
RhythmTreeParser	(class in abjad.tools.rhythmtreetools.RhythmTreeParser.RhythmTreeParser), 796	SchemeParserFinishedException	(class in abjad.tools.exceptiontools.SchemeParserFinishedException), 1672
RomanNumeral	(class in abjad.tools.tonalanalysistools.RomanNumeral.RomanNumeral), 1297	SchemeVector	(class in abjad.tools.schemetools.SchemeVector.SchemeVector), 811
		SchemeVectorConstant	(class in abjad.tools.schemetools.SchemeVectorConstant.SchemeVectorConstant), 811



813  
 Score (class in abjad.tools.scoretools.Score.Score), 849  
 ScoreBlock (class in ab- jad.tools.lilypondfiletools.ScoreBlock.ScoreBlock), 361  
 ScoreMutationAgent (class in ab- jad.tools.mutationtools.ScoreMutationAgent.ScoreMutationAgent), 457  
 SearchTree (class in ab- jad.tools.quantizationtools.SearchTree.SearchTree), 632  
 Segment (class in ab- jad.tools.pitchtools.Segment.Segment), 510  
 select() (in module abjad.tools.selectiontools.select), 891  
 select() (in module ab- jad.tools.tonalanalysistools.select), 1307  
 Selection (class in ab- jad.tools.selectiontools.Selection.Selection), 875  
 SelectionInventory (class in ab- jad.tools.selectiontools.SelectionInventory.SelectionInventory), 877  
 SequentialMusic (class in ab- jad.tools.lilypondparsertools.SequentialMusic.SequentialMusic), 1738  
 SerialJobHandler (class in ab- jad.tools.quantizationtools.SerialJobHandler.SerialJobHandler), 696  
 Set (class in abjad.tools.pitchtools.Set.Set), 512  
 set\_always\_format\_time\_signature\_of\_measures\_in\_expr() (in module ab- jad.tools.measuretools.set\_always\_format\_time\_signature\_of\_measures\_in\_expr), 451  
 set\_line\_breaks\_by\_line\_duration() (in module ab- jad.tools.layouttools.set\_line\_breaks\_by\_line\_duration), 317  
 set\_line\_breaks\_cyclically\_by\_line\_duration\_ge() (in module ab- jad.tools.layouttools.set\_line\_breaks\_cyclically\_by\_line\_duration\_ge), 317  
 set\_line\_breaks\_cyclically\_by\_line\_duration\_in\_seconds\_ge() (in module ab- jad.tools.layouttools.set\_line\_breaks\_cyclically\_by\_line\_duration\_in\_seconds\_ge), 318  
 set\_measure\_denominator\_and\_adjust\_numerator() (in module ab- jad.tools.measuretools.set\_measure\_denominator\_and\_adjust\_numerator), 451  
 set\_written\_pitch\_of\_pitched\_components\_in\_expr() (in module ab- jad.tools.pitchtools.set\_written\_pitch\_of\_pitched\_components\_in\_expr), 617  
 shadow\_pitch\_contour\_reservoir() (in module ab- jad.demos.part.shadow\_pitch\_contour\_reservoir), 1379  
 ShortHairpinCheck (class in ab- jad.tools.wellformednesstools.ShortHairpinCheck.ShortHairpinCheck), 1367  
 show() (in module abjad.tools.iotools.show), 280  
 Sieve (class in abjad.tools.sievetools.Sieve.Sieve), 936  
 sign() (in module abjad.tools.mathtools.sign), 433  
 SilentQEvent (class in ab- jad.tools.quantizationtools.SilentQEvent.SilentQEvent), 697  
 SimpleInequality (class in ab- jad.tools.timerelationtools.SimpleInequality.SimpleInequality), 1195  
 SimultaneousMusic (class in ab- jad.tools.lilypondparsertools.SimultaneousMusic.SimultaneousMusic), 1692  
 SimultaneousSelection (class in ab- jad.tools.selectiontools.SimultaneousSelection.SimultaneousSelection), 881  
 Skip (class in abjad.tools.skiptools.Skip.Skip), 939  
 SkipRhythmMaker (class in ab- jad.tools.rhythm makertools.SkipRhythmMaker.SkipRhythmMaker), 763  
 SliceSelection (class in ab- jad.tools.selectiontools.SliceSelection.SliceSelection), 883  
 SlurSpanner (class in ab- jad.tools.spannertools.SlurSpanner.SlurSpanner), 1069  
 snake\_case\_to\_lower\_camel\_case() (in module ab- jad.tools.stringtools.snake\_case\_to\_lower\_camel\_case), 1127  
 snake\_case\_to\_upper\_camel\_case() (in module ab- jad.tools.stringtools.snake\_case\_to\_upper\_camel\_case), 1127  
 SopranoSaxophone (class in ab- jad.tools.instrumenttools.SopranoSaxophone.SopranoSaxophone), 227  
 SopranoSaxophone (class in ab- jad.tools.instrumenttools.SopranoSaxophone.SopranoSaxophone), 230  
 SopranoVoice (class in ab- jad.tools.instrumenttools.SopranoVoice.SopranoVoice), 233  
 sort\_named\_pitch\_carriers\_in\_expr() (in module ab- jad.tools.pitchtools.sort\_named\_pitch\_carriers\_in\_expr), 477  
 SortedCollection (class in ab- jad.tools.datastructuretools.SortedCollection.SortedCollection), 1452  
 space\_delimited\_lowercase\_to\_upper\_camel\_case() (in module ab- jad.tools.stringtools.space\_delimited\_lowercase\_to\_upper\_camel\_case), 1127  
 SpacingIndication (class in ab- jad.tools.layouttools.SpacingIndication.SpacingIndication), 315  
 Spanner (class in ab- jad.tools.spannertools.Spanner.Spanner), 949

SpannerPopulationError (class in abjad.tools.exceptiontools.SpannerPopulationError), 1673	SuspensionIndicator (class in abjad.tools.tonalanalysis.tools.SuspensionIndicator.SuspensionIndicator), 1303
spawn_subprocess() (in module abjad.tools.iotools.spawn_subprocess), 280	SvnAddAllScript (class in abjad.tools.developerscripttools.SvnAddAllScript.SvnAddAllScript), 1509
spell_numbered_interval_number() (in module abjad.tools.pitchtools.spell_numbered_interval_number), 617	SvnCommitScript (class in abjad.tools.developerscripttools.SvnCommitScript.SvnCommitScript), 1511
spell_pitch_number() (in module abjad.tools.pitchtools.spell_pitch_number), 618	SvnMessageScript (class in abjad.tools.developerscripttools.SvnMessageScript.SvnMessageScript), 1513
splice_new_elements_between_sequence_elements() (in module abjad.tools.sequencetools.splice_new_elements_between_sequence_elements), 923	SvnUpdateScript (class in abjad.tools.developerscripttools.SvnUpdateScript.SvnUpdateScript), 1515
split_sequence_by_weights() (in module abjad.tools.sequencetools.split_sequence_by_weights), 924	SyntaxNode (class in abjad.tools.lilypondparsertools.SyntaxNode.SyntaxNode), 1739
split_sequence_extended_to_weights() (in module abjad.tools.sequencetools.split_sequence_extended_to_weights), 925	T
Staff (class in abjad.tools.stafftools.Staff.Staff), 1114	TaleaRhythmMaker (class in abjad.tools.rhythm makertools.TaleaRhythmMaker.TaleaRhythmMaker), 766
StaffChangeMark (class in abjad.tools.contexttools.StaffChangeMark.StaffChangeMark), 61	TempoError (class in abjad.tools.exceptiontools.TempoError), 1674
StaffGroup (class in abjad.tools.scoretools.StaffGroup.StaffGroup), 857	TempoMark (class in abjad.tools.contexttools.TempoMark.TempoMark), 64
StaffLinesSpanner (class in abjad.tools.spannertools.StaffLinesSpanner.StaffLinesSpanner), 1075	TempoMarkInventory (class in abjad.tools.contexttools.TempoMarkInventory.TempoMarkInventory), 68
StemTremolo (class in abjad.tools.marktools.StemTremolo.StemTremolo), 383	TenorSaxophone (class in abjad.tools.instrumenttools.TenorSaxophone.TenorSaxophone), 236
string_to_accent_free_snake_case() (in module abjad.tools.stringtools.string_to_accent_free_snake_case), 1127	TenorTrombone (class in abjad.tools.instrumenttools.TenorTrombone.TenorTrombone), 239
string_to_space_delimited_lowercase() (in module abjad.tools.stringtools.string_to_space_delimited_lowercase), 1128	TenorVoice (class in abjad.tools.instrumenttools.TenorVoice.TenorVoice), 242
StringOrchestraScoreTemplate (class in abjad.tools.scoretemplatetools.StringOrchestraScoreTemplate.StringOrchestraScoreTemplate), 818	TerminalQEvent (class in abjad.tools.quantizationtools.TerminalQEvent.TerminalQEvent), 869
StringQuartetScoreTemplate (class in abjad.tools.scoretemplatetools.StringQuartetScoreTemplate.StringQuartetScoreTemplate), 820	TestAndRebuildScript (class in abjad.tools.developerscripttools.TestAndRebuildScript.TestAndRebuildScript), 1517
strip_diacritics_from_binary_string() (in module abjad.tools.stringtools.strip_diacritics_from_binary_string), 1128	TextScriptSpanner (class in abjad.tools.spannertools.TextScriptSpanner.TextScriptSpanner), 1081
suggest_clef_for_named_pitches() (in module abjad.tools.pitchtools.suggest_clef_for_named_pitches), 618	TextSpanner (class in abjad.tools.spannertools.TextSpanner.TextSpanner), 1087
sum_consecutive_sequence_elements_by_sign() (in module abjad.tools.sequencetools.sum_consecutive_sequence_elements_by_sign), 925	TieChain (class in abjad.tools.selectiontools.TieChain.TieChain), 885
sum_sequence_elements_at_indices() (in module abjad.tools.sequencetools.sum_sequence_elements_at_indices), 885	TieChainError (class in ab-

jad.tools.exceptiontools.TieChainError), 1675	jad.tools.timerelationtools.timespan_2_overlaps_all_of_timespan_1() 1208
TieSpanner (class in ab-jad.tools.spannertools.TieSpanner.TieSpanner), 1093	timespan_2_overlaps_only_start_of_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_overlaps_only_start_of_timespan_1() 1208
TimeInterval (class in ab-jad.tools.timeintervaltools.TimeInterval.TimeInterval), 1141	timespan_2_overlaps_only_stop_of_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_overlaps_only_stop_of_timespan_1() 1208
TimeIntervalAggregateMixin (class in ab-jad.tools.timeintervaltools.TimeIntervalAggregateMixin.TimeIntervalAggregateMixin), 1133	timespan_2_overlaps_start_of_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_overlaps_start_of_timespan_1() 1209
TimeIntervalMixin (class in ab-jad.tools.timeintervaltools.TimeIntervalMixin.TimeIntervalMixin), 1139	timespan_2_overlaps_stop_of_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_overlaps_stop_of_timespan_1() 1209
TimeIntervalTree (class in ab-jad.tools.timeintervaltools.TimeIntervalTree.TimeIntervalTree), 1144	timespan_2_starts_after_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_starts_after_timespan_1() 1210
TimeIntervalTreeDictionary (class in ab-jad.tools.timeintervaltools.TimeIntervalTreeDictionary.TimeIntervalTreeDictionary), 1160	timespan_2_starts_before_timespan_1_starts() (in module ab-jad.tools.timerelationtools.timespan_2_starts_before_timespan_1_starts() 1210
TimeIntervalTreeNode (class in ab-jad.tools.timeintervaltools.TimeIntervalTreeNode.TimeIntervalTreeNode), 1182	timespan_2_starts_during_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_starts_during_timespan_1() 1211
TimeRelation (class in ab-jad.tools.timerelationtools.TimeRelation.TimeRelation), 1185	timespan_2_starts_when_timespan_1_starts() (in module ab-jad.tools.timerelationtools.timespan_2_starts_when_timespan_1_starts() 1212
TimeSignatureError (class in ab-jad.tools.exceptiontools.TimeSignatureError), 1676	timespan_2_stops_after_timespan_1_starts() (in module ab-jad.tools.timerelationtools.timespan_2_stops_after_timespan_1_starts() 1212
TimeSignatureMark (class in ab-jad.tools.contexttools.TimeSignatureMark.TimeSignatureMark), 72	timespan_2_stops_after_timespan_1_stops() (in module ab-jad.tools.timerelationtools.timespan_2_stops_after_timespan_1_stops() 1212
Timespan (class in ab-jad.tools.timespantools.Timespan.Timespan), 1235	timespan_2_stops_before_timespan_1_starts() (in module ab-jad.tools.timerelationtools.timespan_2_stops_before_timespan_1_starts() 1213
timespan_2_contains_timespan_1_improperly() (in module ab-jad.tools.timerelationtools.timespan_2_contains_timespan_1_improperly), 1205	timespan_2_stops_before_timespan_1_stops() (in module ab-jad.tools.timerelationtools.timespan_2_stops_before_timespan_1_stops() 1213
timespan_2_curtails_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_curtails_timespan_1), 1206	timespan_2_stops_during_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_stops_during_timespan_1() 1213
timespan_2_delays_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_delays_timespan_1), 1206	timespan_2_stops_when_timespan_1_starts() (in module ab-jad.tools.timerelationtools.timespan_2_stops_when_timespan_1_starts() 1212
timespan_2_happens_during_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_happens_during_timespan_1), 1206	timespan_2_stops_when_timespan_1_stops() (in module ab-jad.tools.timerelationtools.timespan_2_stops_when_timespan_1_stops() 1212
timespan_2_intersects_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_intersects_timespan_1), 1207	timespan_2_stops_during_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_stops_during_timespan_1() 1213
timespan_2_is_congruent_to_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_is_congruent_to_timespan_1), 1207	timespan_2_stops_during_timespan_1_stops() (in module ab-jad.tools.timerelationtools.timespan_2_stops_during_timespan_1_stops() 1213
timespan_2_overlaps_all_of_timespan_1() (in module ab-jad.tools.timerelationtools.timespan_2_overlaps_all_of_timespan_1), 1208	timespan_2_stops_during_timespan_1_stops() (in module ab-jad.tools.timerelationtools.timespan_2_stops_during_timespan_1_stops() 1213



(in module ab- 619  
 jad.tools.timerelationtools.timespan\_2\_stops\_before\_timespan\_1\_stops)(class in ab-  
 1213 jad.tools.datastructuretools.TreeContainer.TreeContainer),  
 timespan\_2\_stops\_during\_timespan\_1() (in module ab- 1454  
 1213 jad.tools.timerelationtools.timespan\_2\_stops\_during\_timespan\_1), (class in ab-  
 1213 jad.tools.datastructuretools.TreeNode.TreeNode),  
 timespan\_2\_stops\_when\_timespan\_1\_starts() 1463  
 (in module ab- TrillSpanner (class in ab-  
 jad.tools.timerelationtools.timespan\_2\_stops\_when\_timespan\_1\_starts)ner.tools.TrillSpanner.TrillSpanner),  
 1214 1099  
 timespan\_2\_stops\_when\_timespan\_1\_stops() Trumpet (class in ab-  
 (in module ab- jad.tools.instrumenttools.Trumpet.Trumpet),  
 jad.tools.timerelationtools.timespan\_2\_stops\_when\_timespan\_1\_stops),  
 1214 345  
 timespan\_2\_trisects\_timespan\_1() (in module ab- truncate\_runs\_in\_sequence() (in module ab-  
 jad.tools.timerelationtools.timespan\_2\_trisects\_timespan\_1)jad.tools.sequencetools.truncate\_runs\_in\_sequence),  
 1214 926  
 TimespanInventory (class in ab- truncate\_sequence\_to\_sum() (in module ab-  
 jad.tools.timespantools.TimespanInventory.TimespanInventory),  
 1255 607  
 TimespanTimespanTimeRelation (class in ab- truncate\_sequence\_to\_weight() (in module ab-  
 jad.tools.timerelationtools.TimespanTimespanTimeRelation)jad.tools.sequencetools.truncate\_sequence\_to\_weight),  
 1197 627 TimespanTimespanTimeRelation),  
 TonalAnalysisAgent (class in ab- 248  
 jad.tools.tonalanalysistools.TonalAnalysisAgentTonalAnalysisAgent),  
 1304 1326  
 TonalHarmonyError (class in ab- TupletFuseError (class in ab-  
 jad.tools.exceptiontools.TonalHarmonyError), jad.tools.exceptiontools.TupletFuseError),  
 1677 1678  
 ToolsPackageDocumenter (class in ab- TupletMonadRhythmMaker (class in ab-  
 jad.tools.documentationtools.ToolsPackageDocumenter.ToolsPackageDocumenter),  
 1649 769  
 transpose\_from\_sounding\_pitch\_to\_written\_pitch() TwelveToneRow (class in ab-  
 (in module ab- jad.tools.pitchtools.TwelveToneRow.TwelveToneRow),  
 jad.tools.instrumenttools.transpose\_from\_sounding\_pitch\_to\_written\_pitch),  
 271 605  
 transpose\_from\_written\_pitch\_to\_sounding\_pitch() TwoStaffPianoScoreTemplate (class in ab-  
 (in module ab- jad.tools.scoretemplatetools.TwoStaffPianoScoreTemplate.Two  
 271 821  
 jad.tools.instrumenttools.transpose\_from\_written\_pitch\_to\_sounding\_pitch)(class in ab-  
 271 jad.tools.datastructuretools.TypedCollection.TypedCollection),  
 transpose\_named\_pitch\_by\_numbered\_interval\_and\_respell() 1407  
 (in module ab- TypedCounter (class in ab-  
 jad.tools.pitchtools.transpose\_named\_pitch\_by\_numbered\_interval\_and\_respell)tools.TypedCounter.TypedCounter),  
 618 1466  
 transpose\_pitch\_carrier\_by\_interval() (in module ab- TypedFrozenSet (class in ab-  
 jad.tools.pitchtools.transpose\_pitch\_carrier\_by\_interval), jad.tools.datastructuretools.TypedFrozenSet.TypedFrozenSet),  
 618 1468  
 transpose\_pitch\_class\_number\_to\_pitch\_number\_neighbor() TypedList (class in ab-  
 (in module ab- jad.tools.datastructuretools.TypedList.TypedList),  
 jad.tools.pitchtools.transpose\_pitch\_class\_number\_to\_pitch\_number\_neighbor),  
 619 476  
 transpose\_pitch\_expr\_into\_pitch\_range() TypedTuple (class in ab-  
 (in module ab- jad.tools.datastructuretools.TypedTuple.TypedTuple),  
 jad.tools.pitchtools.transpose\_pitch\_expr\_into\_pitch\_range),  
 619 1475  
 transpose\_pitch\_number\_by\_octave\_transposition\_mapping() UnboundedTimeIntervalError (class in ab-  
 (in module ab- jad.tools.exceptiontools.UnboundedTimeIntervalError),  
 jad.tools.pitchtools.transpose\_pitch\_number\_by\_octave\_transposition\_mapping),  
 619 1659

UnderfullContainerError (class in abjad.tools.exceptiontools.UnderfullContainerError), 1680	X Xylophone (class in abjad.tools.instrumenttools.Xylophone.Xylophone), 267
UntunedPercussion (class in abjad.tools.instrumenttools.UntunedPercussion.UntunedPercussion), 251	Y
UnweightedSearchTree (class in abjad.tools.quantizationtools.UnweightedSearchTree.UnweightedSearchTree), 700	yield_all_combinations_of_sequence_elements() (in module abjad.tools.sequencetools.yield_all_combinations_of_sequence_elements), 927
UpdateManager (class in abjad.tools.updatetools.UpdateManager.UpdateManager), 1755	yield_all_compositions_of_integer() (in module abjad.tools.mathtools.yield_all_compositions_of_integer), 433
upper_camel_case_to_snake_case() (in module abjad.tools.stringtools.upper_camel_case_to_snake_case), 1128	yield_all_k_ary_sequences_of_length() (in module abjad.tools.sequencetools.yield_all_k_ary_sequences_of_length), 928
upper_camel_case_to_space_delimited_lowercase() (in module abjad.tools.stringtools.upper_camel_case_to_space_delimited_lowercase), 1129	yield_all_pairs_between_sequences() (in module abjad.tools.sequencetools.yield_all_pairs_between_sequences), 928
V	yield_all_partitions_of_integer() (in module abjad.tools.mathtools.yield_all_partitions_of_integer), 434
Vector (class in abjad.tools.pitchtools.Vector.Vector), 514	yield_all_partitions_of_sequence() (in module abjad.tools.sequencetools.yield_all_partitions_of_sequence), 928
verify_output_directory() (in module abjad.tools.iotools.verify_output_directory), 280	yield_all_permutations_of_sequence() (in module abjad.tools.sequencetools.yield_all_permutations_of_sequence), 928
VerticalMoment (class in abjad.tools.selectiontools.VerticalMoment.VerticalMoment), 889	yield_all_permutations_of_sequence_in_orbit() (in module abjad.tools.sequencetools.yield_all_permutations_of_sequence_in_orbit), 929
Vibraphone (class in abjad.tools.instrumenttools.Vibraphone.Vibraphone), 254	yield_all_restricted_growth_functions_of_length() (in module abjad.tools.sequencetools.yield_all_restricted_growth_functions_of_length), 929
Viola (class in abjad.tools.instrumenttools.Viola.Viola), 257	yield_all_rotations_of_sequence() (in module abjad.tools.sequencetools.yield_all_rotations_of_sequence), 929
Violin (class in abjad.tools.instrumenttools.Violin.Violin), 260	yield_all_set_partitions_of_sequence() (in module abjad.tools.sequencetools.yield_all_set_partitions_of_sequence), 929
Voice (class in abjad.tools.voicetools.Voice.Voice), 1343	yield_all_subsequences_of_sequence() (in module abjad.tools.sequencetools.yield_all_subsequences_of_sequence), 930
W	yield_all_unordered_pairs_of_sequence() (in module abjad.tools.sequencetools.yield_all_unordered_pairs_of_sequence), 930
warn_almost_full() (in module abjad.tools.iotools.warn_almost_full), 280	yield_nonreduced_fractions() (in module abjad.tools.mathtools.yield_nonreduced_fractions), 434
weight() (in module abjad.tools.mathtools.weight), 433	yield_outer_product_of_sequences() (in module abjad.tools.sequencetools.yield_outer_product_of_sequences), 931
WeightedSearchTree (class in abjad.tools.quantizationtools.WeightedSearchTree.WeightedSearchTree), 702	Z
which() (in module abjad.tools.iotools.which), 281	z() (in module abjad.tools.iotools.z), 281
WoodwindFingering (class in abjad.tools.instrumenttools.WoodwindFingering.WoodwindFingering), 263	
write_expr_to_ly() (in module abjad.tools.iotools.write_expr_to_ly), 281	
write_expr_to_pdf() (in module abjad.tools.iotools.write_expr_to_pdf), 281	
write_test_output() (in module abjad.tools.testtools.write_test_output), 1753	

`zip_sequences_cyclically()` (in module `abjad.tools.sequencetools.zip_sequences_cyclically`),  
[931](#)

`zip_sequences_without_truncation()` (in module `abjad.tools.sequencetools.zip_sequences_without_truncation`),  
[932](#)