
Abjad API

Release 2.14

Trevor Bača, Josiah Wolf Oberholtzer, Víctor Adán

December 19, 2013

I	Core composition packages	1
1	agenttools	3
1.1	Concrete classes	3
1.1.1	agenttools.InspectionAgent	3
1.1.2	agenttools.IterationAgent	9
1.1.3	agenttools.MutationAgent	18
1.1.4	agenttools.PersistenceAgent	33
2	datastructuretools	35
2.1	Abstract classes	35
2.1.1	datastructuretools.TypedCollection	35
2.2	Concrete classes	37
2.2.1	datastructuretools.CyclicList	37
2.2.2	datastructuretools.CyclicMatrix	40
2.2.3	datastructuretools.CyclicPayloadTree	43
2.2.4	datastructuretools.CyclicTuple	57
2.2.5	datastructuretools.Matrix	59
2.2.6	datastructuretools.OrdinalConstant	61
2.2.7	datastructuretools.PayloadTree	63
2.2.8	datastructuretools.SortedCollection	76
2.2.9	datastructuretools.StatalServer	79
2.2.10	datastructuretools.StatalServerCursor	80
2.2.11	datastructuretools.TreeContainer	82
2.2.12	datastructuretools.TreeNode	91
2.2.13	datastructuretools.TypedCounter	95
2.2.14	datastructuretools.TypedFrozenSet	99
2.2.15	datastructuretools.TypedList	102
2.2.16	datastructuretools.TypedTuple	107
3	durationtools	111
3.1	Concrete classes	111
3.1.1	durationtools.Duration	111
3.1.2	durationtools.Multiplier	123
3.1.3	durationtools.Offset	134
4	indicatortools	147
4.1	Concrete classes	147
4.1.1	indicatortools.Annotation	147
4.1.2	indicatortools.Articulation	149
4.1.3	indicatortools.BarLine	151
4.1.4	indicatortools.BendAfter	152
4.1.5	indicatortools.Clef	154
4.1.6	indicatortools.ClefInventory	156
4.1.7	indicatortools.Dynamic	161
4.1.8	indicatortools.IndicatorExpression	163

4.1.9	indicatortools.IsAtSoundingPitch	164
4.1.10	indicatortools.IsUnpitched	165
4.1.11	indicatortools.KeySignature	166
4.1.12	indicatortools.LilyPondCommand	168
4.1.13	indicatortools.LilyPondComment	169
4.1.14	indicatortools.StaffChange	171
4.1.15	indicatortools.StemTremolo	172
4.1.16	indicatortools.Tempo	174
4.1.17	indicatortools.TempoInventory	179
4.1.18	indicatortools.TimeSignature	183
5	instrumenttools	187
5.1	Concrete classes	187
5.1.1	instrumenttools.Accordion	187
5.1.2	instrumenttools.AltoFlute	191
5.1.3	instrumenttools.AltoSaxophone	194
5.1.4	instrumenttools.AltoTrombone	197
5.1.5	instrumenttools.AltoVoice	200
5.1.6	instrumenttools.BaritoneSaxophone	203
5.1.7	instrumenttools.BaritoneVoice	206
5.1.8	instrumenttools.BassClarinet	209
5.1.9	instrumenttools.BassFlute	212
5.1.10	instrumenttools.BassSaxophone	215
5.1.11	instrumenttools.BassTrombone	218
5.1.12	instrumenttools.BassVoice	221
5.1.13	instrumenttools.Bassoon	224
5.1.14	instrumenttools.Cello	227
5.1.15	instrumenttools.ClarinetInA	230
5.1.16	instrumenttools.ClarinetInBFlat	233
5.1.17	instrumenttools.ClarinetInEFlat	236
5.1.18	instrumenttools.Contrabass	239
5.1.19	instrumenttools.ContrabassClarinet	242
5.1.20	instrumenttools.ContrabassFlute	245
5.1.21	instrumenttools.ContrabassSaxophone	248
5.1.22	instrumenttools.Contrabassoon	251
5.1.23	instrumenttools.EnglishHorn	254
5.1.24	instrumenttools.Flute	257
5.1.25	instrumenttools.FrenchHorn	260
5.1.26	instrumenttools.Glockenspiel	263
5.1.27	instrumenttools.Guitar	266
5.1.28	instrumenttools.Harp	269
5.1.29	instrumenttools.Harpsichord	272
5.1.30	instrumenttools.Instrument	275
5.1.31	instrumenttools.InstrumentInventory	277
5.1.32	instrumenttools.InstrumentationSpecifier	281
5.1.33	instrumenttools.Marimba	285
5.1.34	instrumenttools.MezzoSopranoVoice	288
5.1.35	instrumenttools.Oboe	291
5.1.36	instrumenttools.Performer	294
5.1.37	instrumenttools.PerformerInventory	301
5.1.38	instrumenttools.Piano	305
5.1.39	instrumenttools.Piccolo	308
5.1.40	instrumenttools.SopraninoSaxophone	311
5.1.41	instrumenttools.SopranoSaxophone	314
5.1.42	instrumenttools.SopranoVoice	317
5.1.43	instrumenttools.TenorSaxophone	320
5.1.44	instrumenttools.TenorTrombone	323
5.1.45	instrumenttools.TenorVoice	326

5.1.46	instrumenttools.Trumpet	329
5.1.47	instrumenttools.Tuba	332
5.1.48	instrumenttools.UntunedPercussion	335
5.1.49	instrumenttools.Vibraphone	338
5.1.50	instrumenttools.Viola	341
5.1.51	instrumenttools.Violin	344
5.1.52	instrumenttools.WoodwindFingering	347
5.1.53	instrumenttools.Xylophone	351
5.2	Functions	353
5.2.1	instrumenttools.iterate_out_of_range_notes_and_chords	353
5.2.2	instrumenttools.notes_and_chords_are_in_range	354
5.2.3	instrumenttools.notes_and_chords_are_on_expected_clefs	354
5.2.4	instrumenttools.transpose_from_sounding_pitch_to_written_pitch	355
5.2.5	instrumenttools.transpose_from_written_pitch_to_sounding_pitch	355
6	labeltools	357
6.1	Functions	357
6.1.1	labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map	357
6.1.2	labeltools.color_contents_of_container	357
6.1.3	labeltools.color_leaf	358
6.1.4	labeltools.color_leaves_in_expr	358
6.1.5	labeltools.color_measure	359
6.1.6	labeltools.color_measures_with_non_power_of_two_denominators_in_expr	359
6.1.7	labeltools.color_note_head_by_numbered_pitch_class_color_map	360
6.1.8	labeltools.label_leaves_in_expr_with_leaf_depth	360
6.1.9	labeltools.label_leaves_in_expr_with_leaf_duration	361
6.1.10	labeltools.label_leaves_in_expr_with_leaf_durations	361
6.1.11	labeltools.label_leaves_in_expr_with_leaf_indices	362
6.1.12	labeltools.label_leaves_in_expr_with_leaf_numbers	362
6.1.13	labeltools.label_leaves_in_expr_with_named_interval_classes	363
6.1.14	labeltools.label_leaves_in_expr_with_named_intervals	363
6.1.15	labeltools.label_leaves_in_expr_with_numbered_interval_classes	363
6.1.16	labeltools.label_leaves_in_expr_with_numbered_intervals	364
6.1.17	labeltools.label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes	364
6.1.18	labeltools.label_leaves_in_expr_with_pitch_class_numbers	364
6.1.19	labeltools.label_leaves_in_expr_with_pitch_numbers	365
6.1.20	labeltools.label_leaves_in_expr_with_tuplet_depth	365
6.1.21	labeltools.label_leaves_in_expr_with_written_leaf_duration	366
6.1.22	labeltools.label_logical_ties_in_expr_with_logical_tie_duration	366
6.1.23	labeltools.label_logical_ties_in_expr_with_logical_tie_durations	367
6.1.24	labeltools.label_logical_ties_in_expr_with_written_logical_tie_duration	367
6.1.25	labeltools.label_notes_in_expr_with_note_indices	367
6.1.26	labeltools.label_vertical_moments_in_expr_with_interval_class_vectors	368
6.1.27	labeltools.label_vertical_moments_in_expr_with_named_intervals	368
6.1.28	labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes	369
6.1.29	labeltools.label_vertical_moments_in_expr_with_numbered_intervals	369
6.1.30	labeltools.label_vertical_moments_in_expr_with_numbered_pitch_classes	370
6.1.31	labeltools.label_vertical_moments_in_expr_with_pitch_numbers	370
6.1.32	labeltools.remove_markup_from_leaves_in_expr	371
7	layouttools	373
7.1	Concrete classes	373
7.1.1	layouttools.SpacingIndication	373
7.2	Functions	374
7.2.1	layouttools.make_spacing_vector	374
7.2.2	layouttools.set_line_breaks_by_line_duration	375
7.2.3	layouttools.set_line_breaks_cyclically_by_line_duration_ge	375
7.2.4	layouttools.set_line_breaks_cyclically_by_line_duration_in_seconds_ge	376

8	lilypondfiletools	377
8.1	Abstract classes	377
8.1.1	lilypondfiletools.AttributedBlock	377
8.1.2	lilypondfiletools.NonattributedBlock	380
8.2	Concrete classes	382
8.2.1	lilypondfiletools.BookBlock	382
8.2.2	lilypondfiletools.BookpartBlock	385
8.2.3	lilypondfiletools.ContextBlock	387
8.2.4	lilypondfiletools.DateTimeToken	390
8.2.5	lilypondfiletools.HeaderBlock	391
8.2.6	lilypondfiletools.LayoutBlock	394
8.2.7	lilypondfiletools.LilyPondDimension	397
8.2.8	lilypondfiletools.LilyPondFile	398
8.2.9	lilypondfiletools.LilyPondLanguageToken	401
8.2.10	lilypondfiletools.LilyPondVersionToken	402
8.2.11	lilypondfiletools.MIDIBlock	403
8.2.12	lilypondfiletools.PaperBlock	406
8.2.13	lilypondfiletools.ScoreBlock	408
8.3	Functions	411
8.3.1	lilypondfiletools.make_basic_lilypond_file	411
8.3.2	lilypondfiletools.make_floating_time_signature_lilypond_file	411
8.3.3	lilypondfiletools.make_time_signature_context_block	411
9	markuptools	413
9.1	Concrete classes	413
9.1.1	markuptools.Markup	413
9.1.2	markuptools.MarkupCommand	415
9.1.3	markuptools.MarkupInventory	417
9.1.4	markuptools.MusicGlyph	421
9.2	Functions	422
9.2.1	markuptools.combine_markup_commands	422
9.2.2	markuptools.make_big_centered_page_number_markup	423
9.2.3	markuptools.make_blank_line_markup	423
9.2.4	markuptools.make_centered_title_markup	423
9.2.5	markuptools.make_vertically_adjusted_composer_markup	424
10	mathtools	425
10.1	Concrete classes	425
10.1.1	mathtools.BoundedObject	425
10.1.2	mathtools.Infinity	427
10.1.3	mathtools.NegativeInfinity	428
10.1.4	mathtools.NonreducedFraction	430
10.1.5	mathtools.NonreducedRatio	438
10.1.6	mathtools.Ratio	440
10.2	Functions	442
10.2.1	mathtools.are_relatively_prime	442
10.2.2	mathtools.arithmetic_mean	442
10.2.3	mathtools.binomial_coefficient	442
10.2.4	mathtools.cumulative_products	443
10.2.5	mathtools.cumulative_signed_weights	443
10.2.6	mathtools.cumulative_sums	443
10.2.7	mathtools.cumulative_sums_pairwise	443
10.2.8	mathtools.difference_series	444
10.2.9	mathtools.divide_number_by_ratio	444
10.2.10	mathtools.divisors	444
10.2.11	mathtools.factors	445
10.2.12	mathtools.fraction_to_proper_fraction	445
10.2.13	mathtools.get_shared_numeric_sign	445

10.2.14	mathtools.greatest_common_divisor	446
10.2.15	mathtools.greatest_multiple_less_equal	446
10.2.16	mathtools.greatest_power_of_two_less_equal	447
10.2.17	mathtools.integer_equivalent_number_to_integer	447
10.2.18	mathtools.integer_to_base_k_tuple	447
10.2.19	mathtools.integer_to_binary_string	448
10.2.20	mathtools.is_assignable_integer	448
10.2.21	mathtools.is_dotted_integer	448
10.2.22	mathtools.is_integer_equivalent_expr	449
10.2.23	mathtools.is_integer_equivalent_number	449
10.2.24	mathtools.is_negative_integer	449
10.2.25	mathtools.is_nonnegative_integer	450
10.2.26	mathtools.is_nonnegative_integer_equivalent_number	450
10.2.27	mathtools.is_nonnegative_integer_power_of_two	450
10.2.28	mathtools.is_positive_integer	451
10.2.29	mathtools.is_positive_integer_equivalent_number	451
10.2.30	mathtools.is_positive_integer_power_of_two	451
10.2.31	mathtools.least_common_multiple	451
10.2.32	mathtools.least_multiple_greater_equal	452
10.2.33	mathtools.least_power_of_two_greater_equal	452
10.2.34	mathtools.next_integer_partition	453
10.2.35	mathtools.partition_integer_by_ratio	453
10.2.36	mathtools.partition_integer_into_canonic_parts	453
10.2.37	mathtools.partition_integer_into_halves	454
10.2.38	mathtools.partition_integer_into_parts_less_than_double	455
10.2.39	mathtools.partition_integer_into_units	455
10.2.40	mathtools.remove_powers_of_two	456
10.2.41	mathtools.sign	456
10.2.42	mathtools.weight	456
10.2.43	mathtools.yield_all_compositions_of_integer	457
10.2.44	mathtools.yield_all_partitions_of_integer	457
10.2.45	mathtools.yield_nonreduced_fractions	457
11	metertools	459
11.1	Concrete classes	459
11.1.1	metertools.Meter	459
11.1.2	metertools.MetricAccentKernel	466
12	pitcharraytools	469
12.1	Concrete classes	469
12.1.1	pitcharraytools.PitchArray	469
12.1.2	pitcharraytools.PitchArrayCell	475
12.1.3	pitcharraytools.PitchArrayColumn	478
12.1.4	pitcharraytools.PitchArrayInventory	481
12.1.5	pitcharraytools.PitchArrayRow	486
13	pitchtools	491
13.1	Abstract classes	491
13.1.1	pitchtools.Interval	491
13.1.2	pitchtools.IntervalClass	494
13.1.3	pitchtools.Pitch	496
13.1.4	pitchtools.PitchClass	500
13.1.5	pitchtools.Segment	503
13.1.6	pitchtools.Set	506
13.1.7	pitchtools.Vector	509
13.2	Concrete classes	512
13.2.1	pitchtools.Accidental	512
13.2.2	pitchtools.IntervalClassSegment	515
13.2.3	pitchtools.IntervalClassSet	519

13.2.4	pitchtools.IntervalClassVector	523
13.2.5	pitchtools.IntervalSegment	527
13.2.6	pitchtools.IntervalSet	531
13.2.7	pitchtools.IntervalVector	535
13.2.8	pitchtools.NamedInterval	539
13.2.9	pitchtools.NamedIntervalClass	544
13.2.10	pitchtools.NamedInversionEquivalentIntervalClass	546
13.2.11	pitchtools.NamedPitch	548
13.2.12	pitchtools.NamedPitchClass	555
13.2.13	pitchtools.NumberedInterval	560
13.2.14	pitchtools.NumberedIntervalClass	563
13.2.15	pitchtools.NumberedInversionEquivalentIntervalClass	565
13.2.16	pitchtools.NumberedPitch	567
13.2.17	pitchtools.NumberedPitchClass	573
13.2.18	pitchtools.NumberedPitchClassColorMap	578
13.2.19	pitchtools.Octave	580
13.2.20	pitchtools.OctaveTranspositionMapping	583
13.2.21	pitchtools.OctaveTranspositionMappingComponent	588
13.2.22	pitchtools.OctaveTranspositionMappingInventory	589
13.2.23	pitchtools.PitchClassSegment	594
13.2.24	pitchtools.PitchClassSet	599
13.2.25	pitchtools.PitchClassVector	604
13.2.26	pitchtools.PitchRange	607
13.2.27	pitchtools.PitchRangeInventory	611
13.2.28	pitchtools.PitchSegment	615
13.2.29	pitchtools.PitchSet	621
13.2.30	pitchtools.PitchVector	625
13.2.31	pitchtools.TwelveToneRow	629
13.3	Functions	633
13.3.1	pitchtools.apply_accidental_to_named_pitch	633
13.3.2	pitchtools.clef_and_staff_position_number_to_named_pitch	633
13.3.3	pitchtools.contains_subsegment	634
13.3.4	pitchtools.get_named_pitch_from_pitch_carrier	634
13.3.5	pitchtools.get_numbered_pitch_class_from_pitch_carrier	635
13.3.6	pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list	635
13.3.7	pitchtools.instantiate_pitch_and_interval_test_collection	636
13.3.8	pitchtools.inventory_aggregate_subsets	636
13.3.9	pitchtools.iterate_named_pitch_pairs_in_expr	636
13.3.10	pitchtools.list_named_pitches_in_expr	637
13.3.11	pitchtools.list_numbered_interval_numbers_pairwise	638
13.3.12	pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise	638
13.3.13	pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range	639
13.3.14	pitchtools.list_ordered_named_pitch_pairs_from_expr_1_to_expr_2	640
13.3.15	pitchtools.list_pitch_numbers_in_expr	640
13.3.16	pitchtools.list_unordered_named_pitch_pairs_in_expr	640
13.3.17	pitchtools.make_n_middle_c_centered_pitches	640
13.3.18	pitchtools.named_pitch_and_clef_to_staff_position_number	641
13.3.19	pitchtools.numbered_inversion_equivalent_interval_class_dictionary	641
13.3.20	pitchtools.permute_named_pitch_carrier_list_by_twelve_tone_row	641
13.3.21	pitchtools.register_pitch_class_numbers_by_pitch_number_aggregate	642
13.3.22	pitchtools.set_written_pitch_of_pitched_components_in_expr	642
13.3.23	pitchtools.sort_named_pitch_carriers_in_expr	642
13.3.24	pitchtools.spell_numbered_interval_number	642
13.3.25	pitchtools.spell_pitch_number	642
13.3.26	pitchtools.suggest_clef_for_named_pitches	643
13.3.27	pitchtools.transpose_named_pitch_by_numbered_interval_and_respell	643
13.3.28	pitchtools.transpose_pitch_carrier_by_interval	643
13.3.29	pitchtools.transpose_pitch_class_number_to_pitch_number_neighbor	643

13.3.30	pitchtools.transpose_pitch_expr_into_pitch_range	644
13.3.31	pitchtools.transpose_pitch_number_by_octave_transposition_mapping	644
14	quantizationtools	647
14.1	Abstract classes	647
14.1.1	quantizationtools.AttackPointOptimizer	647
14.1.2	quantizationtools.GraceHandler	649
14.1.3	quantizationtools.Heuristic	650
14.1.4	quantizationtools.JobHandler	651
14.1.5	quantizationtools.QEvent	653
14.1.6	quantizationtools.QSchema	654
14.1.7	quantizationtools.QSchemaItem	656
14.1.8	quantizationtools.QTarget	657
14.1.9	quantizationtools.SearchTree	659
14.2	Concrete classes	660
14.2.1	quantizationtools.BeatwiseQSchema	660
14.2.2	quantizationtools.BeatwiseQSchemaItem	665
14.2.3	quantizationtools.BeatwiseQTarget	667
14.2.4	quantizationtools.CollapsingGraceHandler	668
14.2.5	quantizationtools.ConcatenatingGraceHandler	669
14.2.6	quantizationtools.DiscardingGraceHandler	671
14.2.7	quantizationtools.DistanceHeuristic	672
14.2.8	quantizationtools.MeasurewiseAttackPointOptimizer	673
14.2.9	quantizationtools.MeasurewiseQSchema	674
14.2.10	quantizationtools.MeasurewiseQSchemaItem	678
14.2.11	quantizationtools.MeasurewiseQTarget	680
14.2.12	quantizationtools.NaiveAttackPointOptimizer	682
14.2.13	quantizationtools.NullAttackPointOptimizer	683
14.2.14	quantizationtools.ParallelJobHandler	684
14.2.15	quantizationtools.ParallelJobHandlerWorker	685
14.2.16	quantizationtools.PitchedQEvent	687
14.2.17	quantizationtools.QEventProxy	688
14.2.18	quantizationtools.QEventSequence	690
14.2.19	quantizationtools.QGrid	696
14.2.20	quantizationtools.QGridContainer	699
14.2.21	quantizationtools.QGridLeaf	711
14.2.22	quantizationtools.QTargetBeat	717
14.2.23	quantizationtools.QTargetMeasure	719
14.2.24	quantizationtools.QuantizationJob	723
14.2.25	quantizationtools.Quantizer	725
14.2.26	quantizationtools.SerialJobHandler	729
14.2.27	quantizationtools.SilentQEvent	730
14.2.28	quantizationtools.TerminalQEvent	731
14.2.29	quantizationtools.UnweightedSearchTree	733
14.2.30	quantizationtools.WeightedSearchTree	735
14.3	Functions	737
14.3.1	quantizationtools.make_test_time_segments	737
15	rhythmmakertools	739
15.1	Abstract classes	739
15.1.1	rhythmmakertools.BurnishedRhythmMaker	739
15.1.2	rhythmmakertools.DivisionIncisedRhythmMaker	742
15.1.3	rhythmmakertools.IncisedRhythmMaker	745
15.1.4	rhythmmakertools.OutputIncisedRhythmMaker	747
15.1.5	rhythmmakertools.RhythmMaker	749
15.2	Concrete classes	751
15.2.1	rhythmmakertools.DivisionBurnishedTaleaRhythmMaker	751
15.2.2	rhythmmakertools.DivisionIncisedNoteRhythmMaker	756

15.2.3	rhythmmakertools.DivisionIncisedRestRhythmMaker	761
15.2.4	rhythmmakertools.EqualDivisionRhythmMaker	765
15.2.5	rhythmmakertools.EvenRunRhythmMaker	768
15.2.6	rhythmmakertools.NoteRhythmMaker	771
15.2.7	rhythmmakertools.OutputBurnishedTaleaRhythmMaker	774
15.2.8	rhythmmakertools.OutputIncisedNoteRhythmMaker	778
15.2.9	rhythmmakertools.OutputIncisedRestRhythmMaker	781
15.2.10	rhythmmakertools.RestRhythmMaker	785
15.2.11	rhythmmakertools.SkipRhythmMaker	788
15.2.12	rhythmmakertools.TaleaRhythmMaker	790
15.2.13	rhythmmakertools.TupletMonadRhythmMaker	793
16	rhythmtreetools	797
16.1	Abstract classes	797
16.1.1	rhythmtreetools.RhythmTreeNode	797
16.2	Concrete classes	803
16.2.1	rhythmtreetools.RhythmTreeContainer	803
16.2.2	rhythmtreetools.RhythmTreeLeaf	816
16.2.3	rhythmtreetools.RhythmTreeParser	822
16.3	Functions	825
16.3.1	rhythmtreetools.parse_rtm_syntax	825
17	schemetools	827
17.1	Concrete classes	827
17.1.1	schemetools.Scheme	827
17.1.2	schemetools.SchemeAssociativeList	830
17.1.3	schemetools.SchemeColor	832
17.1.4	schemetools.SchemeMoment	834
17.1.5	schemetools.SchemePair	836
17.1.6	schemetools.SchemeVector	838
17.1.7	schemetools.SchemeVectorConstant	840
18	scoretools	843
18.1	Abstract classes	843
18.1.1	scoretools.Component	843
18.1.2	scoretools.Leaf	845
18.2	Concrete classes	847
18.2.1	scoretools.Chord	847
18.2.2	scoretools.Cluster	851
18.2.3	scoretools.Container	857
18.2.4	scoretools.Context	863
18.2.5	scoretools.FixedDurationContainer	870
18.2.6	scoretools.FixedDurationTuplet	876
18.2.7	scoretools.GraceContainer	893
18.2.8	scoretools.GrandStaff	899
18.2.9	scoretools.Measure	906
18.2.10	scoretools.MultimeasureRest	915
18.2.11	scoretools.Note	917
18.2.12	scoretools.NoteHead	919
18.2.13	scoretools.NoteHeadInventory	922
18.2.14	scoretools.PianoStaff	927
18.2.15	scoretools.Rest	934
18.2.16	scoretools.RhythmicStaff	936
18.2.17	scoretools.Score	943
18.2.18	scoretools.Skip	950
18.2.19	scoretools.Staff	952
18.2.20	scoretools.StaffGroup	959
18.2.21	scoretools.Tuplet	966
18.2.22	scoretools.Voice	983

18.3	Functions	989
18.3.1	scoretools.append_spacer_skip_to_underfull_measure	989
18.3.2	scoretools.append_spacer_skips_to_underfull_measures_in_expr	989
18.3.3	scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr	990
18.3.4	scoretools.extend_measures_in_expr_and_apply_full_measure_tuplets	990
18.3.5	scoretools.fill_measures_in_expr_with_full_measure_spacer_skips	990
18.3.6	scoretools.fill_measures_in_expr_with_minimal_number_of_notes	991
18.3.7	scoretools.fill_measures_in_expr_with_repeated_notes	991
18.3.8	scoretools.fill_measures_in_expr_with_time_signature_denominator_notes	991
18.3.9	scoretools.get_measure_that_starts_with_container	991
18.3.10	scoretools.get_measure_that_stops_with_container	991
18.3.11	scoretools.get_next_measure_from_component	992
18.3.12	scoretools.get_one_indexed_measure_number_in_expr	992
18.3.13	scoretools.get_previous_measure_from_component	992
18.3.14	scoretools.make_empty_piano_score	993
18.3.15	scoretools.make_leaves	993
18.3.16	scoretools.make_leaves_from_talea	996
18.3.17	scoretools.make_multimeasure_rests	996
18.3.18	scoretools.make_multiplied_quarter_notes	997
18.3.19	scoretools.make_notes	997
18.3.20	scoretools.make_notes_with_multiplied_durations	998
18.3.21	scoretools.make_percussion_note	998
18.3.22	scoretools.make_piano_score_from_leaves	998
18.3.23	scoretools.make_piano_sketch_score_from_leaves	999
18.3.24	scoretools.make_repeated_notes	999
18.3.25	scoretools.make_repeated_notes_from_time_signature	1000
18.3.26	scoretools.make_repeated_notes_from_time_signatures	1000
18.3.27	scoretools.make_repeated_notes_with_shorter_notes_at_end	1000
18.3.28	scoretools.make_repeated_rests_from_time_signatures	1001
18.3.29	scoretools.make_repeated_skips_from_time_signatures	1001
18.3.30	scoretools.make_rests	1001
18.3.31	scoretools.make_rhythmic_sketch_staff	1001
18.3.32	scoretools.make_skips_with_multiplied_durations	1002
18.3.33	scoretools.make_spacer_skip_measures	1002
18.3.34	scoretools.make_tied_leaf	1002
18.3.35	scoretools.move_full_measure_tuplet_prolation_to_measure_time_signature	1003
18.3.36	scoretools.move_measure_prolation_to_full_measure_tuplet	1003
18.3.37	scoretools.replace_contents_of_measures_in_expr	1003
18.3.38	scoretools.scale_measure_denominator_and_adjust_measure_contents	1004
18.3.39	scoretools.set_always_format_time_signature_of_measures_in_expr	1004
18.3.40	scoretools.set_measure_denominator_and_adjust_numerator	1005
19	selectiontools	1007
19.1	Concrete classes	1007
19.1.1	selectiontools.ContiguousSelection	1007
19.1.2	selectiontools.Descendants	1010
19.1.3	selectiontools.Lineage	1012
19.1.4	selectiontools.LogicalTie	1014
19.1.5	selectiontools.Parentage	1018
19.1.6	selectiontools.Selection	1022
19.1.7	selectiontools.SelectionInventory	1024
19.1.8	selectiontools.SimultaneousSelection	1028
19.1.9	selectiontools.SliceSelection	1030
19.1.10	selectiontools.VerticalMoment	1032
20	sequencetools	1037
20.1	Functions	1037
20.1.1	sequencetools.all_are_assignable_integers	1037

20.1.2	sequencetools.all_are_equal	1037
20.1.3	sequencetools.all_are_integer_equivalent_exprs	1037
20.1.4	sequencetools.all_are_integer_equivalent_numbers	1038
20.1.5	sequencetools.all_are_nonnegative_integer_equivalent_numbers	1038
20.1.6	sequencetools.all_are_nonnegative_integer_powers_of_two	1038
20.1.7	sequencetools.all_are_nonnegative_integers	1038
20.1.8	sequencetools.all_are_numbers	1039
20.1.9	sequencetools.all_are_pairs	1039
20.1.10	sequencetools.all_are_pairs_of_types	1039
20.1.11	sequencetools.all_are_positive_integer_equivalent_numbers	1040
20.1.12	sequencetools.all_are_positive_integers	1040
20.1.13	sequencetools.all_are_unequal	1040
20.1.14	sequencetools.count_length_two_runs_in_sequence	1040
20.1.15	sequencetools.divide_sequence_elements_by_greatest_common_divisor	1041
20.1.16	sequencetools.flatten_sequence	1041
20.1.17	sequencetools.flatten_sequence_at_indices	1041
20.1.18	sequencetools.get_indices_of_sequence_elements_equal_to_true	1042
20.1.19	sequencetools.get_sequence_degree_of_rotational_symmetry	1042
20.1.20	sequencetools.get_sequence_element_at_cyclic_index	1042
20.1.21	sequencetools.get_sequence_elements_at_indices	1043
20.1.22	sequencetools.get_sequence_elements_frequency_distribution	1043
20.1.23	sequencetools.get_sequence_period_of_rotation	1043
20.1.24	sequencetools.increase_sequence_elements_at_indices_by_addenda	1043
20.1.25	sequencetools.increase_sequence_elements_cyclically_by_addenda	1044
20.1.26	sequencetools.interlace_sequences	1044
20.1.27	sequencetools.is_fraction_equivalent_pair	1044
20.1.28	sequencetools.is_integer_equivalent_n_tuple	1044
20.1.29	sequencetools.is_integer_equivalent_pair	1045
20.1.30	sequencetools.is_integer_equivalent_singleton	1045
20.1.31	sequencetools.is_integer_n_tuple	1045
20.1.32	sequencetools.is_integer_pair	1045
20.1.33	sequencetools.is_integer_singleton	1046
20.1.34	sequencetools.is_monotonically_decreasing_sequence	1046
20.1.35	sequencetools.is_monotonically_increasing_sequence	1046
20.1.36	sequencetools.is_n_tuple	1047
20.1.37	sequencetools.is_null_tuple	1047
20.1.38	sequencetools.is_pair	1047
20.1.39	sequencetools.is_permutation	1048
20.1.40	sequencetools.is_repetition_free_sequence	1048
20.1.41	sequencetools.is_restricted_growth_function	1048
20.1.42	sequencetools.is_singleton	1049
20.1.43	sequencetools.is_strictly_decreasing_sequence	1049
20.1.44	sequencetools.is_strictly_increasing_sequence	1050
20.1.45	sequencetools.iterate_sequence_cyclically	1050
20.1.46	sequencetools.iterate_sequence_cyclically_from_start_to_stop	1051
20.1.47	sequencetools.iterate_sequence_forward_and_backward_nonoverlapping	1051
20.1.48	sequencetools.iterate_sequence_forward_and_backward_overlapping	1051
20.1.49	sequencetools.iterate_sequence_nwise_cyclic	1051
20.1.50	sequencetools.iterate_sequence_nwise_strict	1052
20.1.51	sequencetools.iterate_sequence_nwise_wrapped	1052
20.1.52	sequencetools.iterate_sequence_pairwise_cyclic	1052
20.1.53	sequencetools.iterate_sequence_pairwise_strict	1053
20.1.54	sequencetools.iterate_sequence_pairwise_wrapped	1053
20.1.55	sequencetools.join_subsequences	1053
20.1.56	sequencetools.join_subsequences_by_sign_of_subsequence_elements	1053
20.1.57	sequencetools.map_sequence_elements_to_canonic_tuples	1053
20.1.58	sequencetools.map_sequence_elements_to_numbered_sublists	1054
20.1.59	sequencetools.merge_duration_sequences	1054

20.1.60	sequencetools.negate_absolute_value_of_sequence_elements_at_indices	1054
20.1.61	sequencetools.negate_absolute_value_of_sequence_elements_cyclically	1054
20.1.62	sequencetools.negate_sequence_elements_at_indices	1055
20.1.63	sequencetools.negate_sequence_elements_cyclically	1055
20.1.64	sequencetools.override_sequence_elements_at_indices	1055
20.1.65	sequencetools.pair_duration_sequence_elements_with_input_pair_values	1055
20.1.66	sequencetools.partition_sequence_by_backgrounded_weights	1056
20.1.67	sequencetools.partition_sequence_by_counts	1056
20.1.68	sequencetools.partition_sequence_by_ratio_of_lengths	1058
20.1.69	sequencetools.partition_sequence_by_ratio_of_weights	1058
20.1.70	sequencetools.partition_sequence_by_restricted_growth_function	1059
20.1.71	sequencetools.partition_sequence_by_sign_of_elements	1059
20.1.72	sequencetools.partition_sequence_by_value_of_elements	1060
20.1.73	sequencetools.partition_sequence_by_weights_at_least	1060
20.1.74	sequencetools.partition_sequence_by_weights_at_most	1060
20.1.75	sequencetools.partition_sequence_by_weights_exactly	1061
20.1.76	sequencetools.partition_sequence_extended_to_counts	1062
20.1.77	sequencetools.permute_sequence	1062
20.1.78	sequencetools.remap_sequence_by_range_pairs	1062
20.1.79	sequencetools.remove_sequence_elements_at_indices	1063
20.1.80	sequencetools.remove_sequence_elements_at_indices_cyclically	1063
20.1.81	sequencetools.remove_subsequence_of_weight_at_index	1063
20.1.82	sequencetools.repeat_runs_in_sequence_to_count	1063
20.1.83	sequencetools.repeat_sequence_elements_at_indices	1064
20.1.84	sequencetools.repeat_sequence_elements_at_indices_cyclically	1064
20.1.85	sequencetools.repeat_sequence_elements_n_times_each	1065
20.1.86	sequencetools.repeat_sequence_n_times	1065
20.1.87	sequencetools.repeat_sequence_to_length	1065
20.1.88	sequencetools.repeat_sequence_to_weight_at_least	1065
20.1.89	sequencetools.repeat_sequence_to_weight_at_most	1066
20.1.90	sequencetools.repeat_sequence_to_weight_exactly	1066
20.1.91	sequencetools.replace_sequence_elements_cyclically_with_new_material	1066
20.1.92	sequencetools.retain_sequence_elements_at_indices	1066
20.1.93	sequencetools.retain_sequence_elements_at_indices_cyclically	1067
20.1.94	sequencetools.reverse_sequence	1067
20.1.95	sequencetools.reverse_sequence_elements	1067
20.1.96	sequencetools.rotate_sequence	1067
20.1.97	sequencetools.splice_new_elements_between_sequence_elements	1067
20.1.98	sequencetools.split_sequence_by_weights	1068
20.1.99	sequencetools.split_sequence_extended_to_weights	1069
20.1.100	sequencetools.sum_consecutive_sequence_elements_by_sign	1069
20.1.101	sequencetools.sum_sequence_elements_at_indices	1070
20.1.102	sequencetools.truncate_runs_in_sequence	1070
20.1.103	sequencetools.truncate_sequence_to_sum	1070
20.1.104	sequencetools.truncate_sequence_to_weight	1071
20.1.105	sequencetools.yield_all_combinations_of_sequence_elements	1071
20.1.106	sequencetools.yield_all_k_ary_sequences_of_length	1072
20.1.107	sequencetools.yield_all_pairs_between_sequences	1072
20.1.108	sequencetools.yield_all_partitions_of_sequence	1072
20.1.109	sequencetools.yield_all_permutations_of_sequence	1073
20.1.110	sequencetools.yield_all_permutations_of_sequence_in_orbit	1073
20.1.111	sequencetools.yield_all_restricted_growth_functions_of_length	1073
20.1.112	sequencetools.yield_all_rotations_of_sequence	1073
20.1.113	sequencetools.yield_all_set_partitions_of_sequence	1074
20.1.114	sequencetools.yield_all_subsequences_of_sequence	1074
20.1.115	sequencetools.yield_all_unordered_pairs_of_sequence	1075
20.1.116	sequencetools.yield_outer_product_of_sequences	1075
20.1.117	sequencetools.zip_sequences_cyclically	1075

20.1.118	sequencetools.zip_sequences_without_truncation	1076
21	sievetools	1077
21.1	Concrete classes	1077
21.1.1	sievetools.BaseResidueClass	1077
21.1.2	sievetools.ResidueClass	1079
21.1.3	sievetools.Sieve	1081
22	spannertools	1085
22.1	Concrete classes	1085
22.1.1	spannertools.Beam	1085
22.1.2	spannertools.ComplexBeam	1089
22.1.3	spannertools.Crescendo	1093
22.1.4	spannertools.Decrescendo	1098
22.1.5	spannertools.DuratedComplexBeam	1103
22.1.6	spannertools.DynamicTextSpanner	1108
22.1.7	spannertools.Glissando	1111
22.1.8	spannertools.Hairpin	1114
22.1.9	spannertools.HiddenStaffSpanner	1119
22.1.10	spannertools.HorizontalBracketSpanner	1122
22.1.11	spannertools.MeasuredComplexBeam	1125
22.1.12	spannertools.MultipartBeam	1130
22.1.13	spannertools.OctavationSpanner	1133
22.1.14	spannertools.PhrasingSlur	1137
22.1.15	spannertools.PianoPedalSpanner	1140
22.1.16	spannertools.Slur	1143
22.1.17	spannertools.Spanner	1146
22.1.18	spannertools.StaffLinesSpanner	1149
22.1.19	spannertools.TextScriptSpanner	1152
22.1.20	spannertools.TextSpanner	1155
22.1.21	spannertools.Tie	1158
22.1.22	spannertools.TrillSpanner	1161
22.2	Functions	1164
22.2.1	spannertools.make_colored_text_spanner_with_nibs	1164
22.2.2	spannertools.make_dynamic_spanner_below_with_nib_at_right	1164
22.2.3	spannertools.make_solid_text_spanner_with_nib	1164
23	stringtools	1167
23.1	Functions	1167
23.1.1	stringtools.add_terminal_newlines	1167
23.1.2	stringtools.arg_to_bidirectional_direction_string	1167
23.1.3	stringtools.arg_to_bidirectional_lilypond_symbol	1167
23.1.4	stringtools.arg_to_tridirectional_direction_string	1168
23.1.5	stringtools.arg_to_tridirectional_lilypond_symbol	1168
23.1.6	stringtools.arg_to_tridirectional_ordinal_constant	1169
23.1.7	stringtools.capitalize_string_start	1169
23.1.8	stringtools.format_input_lines_as_doc_string	1169
23.1.9	stringtools.format_input_lines_as_regression_test	1170
23.1.10	stringtools.is_dash_case_file_name	1170
23.1.11	stringtools.is_dash_case_string	1171
23.1.12	stringtools.is_lower_camel_case_string	1171
23.1.13	stringtools.is_snake_case_file_name	1171
23.1.14	stringtools.is_snake_case_file_name_with_extension	1171
23.1.15	stringtools.is_snake_case_package_name	1172
23.1.16	stringtools.is_snake_case_string	1172
23.1.17	stringtools.is_space_delimited_lowercase_string	1172
23.1.18	stringtools.is_upper_camel_case_string	1172
23.1.19	stringtools.pluralize_string	1173
23.1.20	stringtools.snake_case_to_lower_camel_case	1173

23.1.21	stringtools.snake_case_to_upper_camel_case	1173
23.1.22	stringtools.space_delimited_lowercase_to_upper_camel_case	1173
23.1.23	stringtools.string_to_accent_free_snake_case	1173
23.1.24	stringtools.string_to_space_delimited_lowercase	1174
23.1.25	stringtools.strip_diacritics_from_binary_string	1174
23.1.26	stringtools.upper_camel_case_to_snake_case	1174
23.1.27	stringtools.upper_camel_case_to_space_delimited_lowercase	1174
24	templatetools	1177
24.1	Concrete classes	1177
24.1.1	templatetools.GroupedRhythmicStavesScoreTemplate	1177
24.1.2	templatetools.GroupedStavesScoreTemplate	1179
24.1.3	templatetools.StringOrchestraScoreTemplate	1181
24.1.4	templatetools.StringQuartetScoreTemplate	1182
24.1.5	templatetools.TwoStaffPianoScoreTemplate	1183
25	timespantools	1185
25.1	Abstract classes	1185
25.1.1	timespantools.TimeRelation	1185
25.2	Concrete classes	1187
25.2.1	timespantools.CompoundInequality	1187
25.2.2	timespantools.OffsetTimespanTimeRelation	1192
25.2.3	timespantools.SimpleInequality	1194
25.2.4	timespantools.Timespan	1196
25.2.5	timespantools.TimespanInventory	1216
25.2.6	timespantools.TimespanTimespanTimeRelation	1246
25.3	Functions	1251
25.3.1	timespantools.offset_happens_after_timespan_starts	1251
25.3.2	timespantools.offset_happens_after_timespan_stops	1251
25.3.3	timespantools.offset_happens_before_timespan_starts	1252
25.3.4	timespantools.offset_happens_before_timespan_stops	1253
25.3.5	timespantools.offset_happens_during_timespan	1253
25.3.6	timespantools.offset_happens_when_timespan_starts	1254
25.3.7	timespantools.offset_happens_when_timespan_stops	1254
25.3.8	timespantools.timespan_2_contains_timespan_1_improperly	1254
25.3.9	timespantools.timespan_2_curtails_timespan_1	1255
25.3.10	timespantools.timespan_2_delays_timespan_1	1255
25.3.11	timespantools.timespan_2_happens_during_timespan_1	1256
25.3.12	timespantools.timespan_2_intersects_timespan_1	1256
25.3.13	timespantools.timespan_2_is_congruent_to_timespan_1	1256
25.3.14	timespantools.timespan_2_overlaps_all_of_timespan_1	1257
25.3.15	timespantools.timespan_2_overlaps_only_start_of_timespan_1	1257
25.3.16	timespantools.timespan_2_overlaps_only_stop_of_timespan_1	1257
25.3.17	timespantools.timespan_2_overlaps_start_of_timespan_1	1258
25.3.18	timespantools.timespan_2_overlaps_stop_of_timespan_1	1258
25.3.19	timespantools.timespan_2_starts_after_timespan_1_starts	1258
25.3.20	timespantools.timespan_2_starts_after_timespan_1_stops	1259
25.3.21	timespantools.timespan_2_starts_before_timespan_1_starts	1259
25.3.22	timespantools.timespan_2_starts_before_timespan_1_stops	1259
25.3.23	timespantools.timespan_2_starts_during_timespan_1	1260
25.3.24	timespantools.timespan_2_starts_when_timespan_1_starts	1261
25.3.25	timespantools.timespan_2_starts_when_timespan_1_stops	1261
25.3.26	timespantools.timespan_2_stops_after_timespan_1_starts	1261
25.3.27	timespantools.timespan_2_stops_after_timespan_1_stops	1262
25.3.28	timespantools.timespan_2_stops_before_timespan_1_starts	1262
25.3.29	timespantools.timespan_2_stops_before_timespan_1_stops	1262
25.3.30	timespantools.timespan_2_stops_during_timespan_1	1263
25.3.31	timespantools.timespan_2_stops_when_timespan_1_starts	1263

25.3.32	timespantools.timespan_2_stops_when_timespan_1_stops	1263
25.3.33	timespantools.timespan_2_trisects_timespan_1	1263
26	tonalanalysistools	1265
26.1	Concrete classes	1265
26.1.1	tonalanalysistools.ChordExtent	1265
26.1.2	tonalanalysistools.ChordInversion	1266
26.1.3	tonalanalysistools.ChordOmission	1268
26.1.4	tonalanalysistools.ChordQuality	1269
26.1.5	tonalanalysistools.ChordSuspension	1270
26.1.6	tonalanalysistools.Mode	1272
26.1.7	tonalanalysistools.RomanNumeral	1273
26.1.8	tonalanalysistools.RootedChordClass	1275
26.1.9	tonalanalysistools.RootlessChordClass	1281
26.1.10	tonalanalysistools.Scale	1286
26.1.11	tonalanalysistools.ScaleDegree	1291
26.1.12	tonalanalysistools.TonalAnalysisAgent	1293
26.2	Functions	1296
26.2.1	tonalanalysistools.select	1296
27	topleveltools	1297
27.1	Functions	1297
27.1.1	topleveltools.attach	1297
27.1.2	topleveltools.contextualize	1297
27.1.3	topleveltools.detach	1297
27.1.4	topleveltools.graph	1297
27.1.5	topleveltools.iterate	1298
27.1.6	topleveltools.mutate	1298
27.1.7	topleveltools.new	1298
27.1.8	topleveltools.override	1299
27.1.9	topleveltools.parse	1299
27.1.10	topleveltools.persist	1299
27.1.11	topleveltools.play	1299
27.1.12	topleveltools.select	1300
27.1.13	topleveltools.show	1300
II	Demos and example packages	1303
28	desordre	1305
28.1	Functions	1305
28.1.1	desordre.make_desordre_cell	1305
28.1.2	desordre.make_desordre_lilypond_file	1305
28.1.3	desordre.make_desordre_measure	1305
28.1.4	desordre.make_desordre_pitches	1305
28.1.5	desordre.make_desordre_score	1305
28.1.6	desordre.make_desordre_staff	1305
29	ferneyhough	1307
29.1	Functions	1307
29.1.1	ferneyhough.configure_lilypond_file	1307
29.1.2	ferneyhough.configure_score	1307
29.1.3	ferneyhough.make_lilypond_file	1307
29.1.4	ferneyhough.make_nested_tuplet	1307
29.1.5	ferneyhough.make_row_of_nested_tuplets	1307
29.1.6	ferneyhough.make_rows_of_nested_tuplets	1307
29.1.7	ferneyhough.make_score	1307
30	mozart	1309

30.1	Functions	1309
30.1.1	mozart.choose_mozart_measures	1309
30.1.2	mozart.make_mozart_lilypond_file	1309
30.1.3	mozart.make_mozart_measure	1309
30.1.4	mozart.make_mozart_measure_corpus	1309
30.1.5	mozart.make_mozart_score	1309
31	part	1311
31.1	Concrete classes	1311
31.1.1	part.PartCantusScoreTemplate	1311
31.2	Functions	1312
31.2.1	part.add_bell_music_to_score	1312
31.2.2	part.add_string_music_to_score	1312
31.2.3	part.apply_bowing_marks	1312
31.2.4	part.apply_dynamics	1312
31.2.5	part.apply_expressive_marks	1312
31.2.6	part.apply_final_bar_lines	1312
31.2.7	part.apply_page_breaks	1312
31.2.8	part.apply_rehearsal_marks	1313
31.2.9	part.configure_lilypond_file	1313
31.2.10	part.configure_score	1313
31.2.11	part.create_pitch_contour_reservoir	1313
31.2.12	part.durate_pitch_contour_reservoir	1313
31.2.13	part.edit_bass_voice	1313
31.2.14	part.edit_cello_voice	1313
31.2.15	part.edit_first_violin_voice	1313
31.2.16	part.edit_second_violin_voice	1313
31.2.17	part.edit_viola_voice	1313
31.2.18	part.make_part_lilypond_file	1314
31.2.19	part.shadow_pitch_contour_reservoir	1314
III	Abjad internal packages	1315
32	abctools	1317
32.1	Abstract classes	1317
32.1.1	abctools.ContextManager	1317
32.1.2	abctools.Parser	1318
32.2	Concrete classes	1320
32.2.1	abctools.AbjadObject	1320
33	abjadbooktools	1321
33.1	Abstract classes	1321
33.1.1	abjadbooktools.OutputFormat	1321
33.2	Concrete classes	1323
33.2.1	abjadbooktools.AbjadBookProcessor	1323
33.2.2	abjadbooktools.AbjadBookScript	1325
33.2.3	abjadbooktools.CodeBlock	1327
33.2.4	abjadbooktools.HTMLOutputFormat	1329
33.2.5	abjadbooktools.LaTeXOutputFormat	1330
33.2.6	abjadbooktools.ReSTOutputFormat	1332
34	developerscripttools	1335
34.1	Abstract classes	1335
34.1.1	developerscripttools.DeveloperScript	1335
34.1.2	developerscripttools.DirectoryScript	1338
34.2	Concrete classes	1340
34.2.1	developerscripttools.AbjDevScript	1340
34.2.2	developerscripttools.AbjGrepScript	1343

34.2.3	developerscripttools.BuildApiScript	1345
34.2.4	developerscripttools.CleanScript	1348
34.2.5	developerscripttools.CountLinewidthsScript	1350
34.2.6	developerscripttools.CountToolsScript	1353
34.2.7	developerscripttools.MakeNewClassTemplateScript	1355
34.2.8	developerscripttools.MakeNewFunctionTemplateScript	1357
34.2.9	developerscripttools.PyTestScript	1360
34.2.10	developerscripttools.RenameModulesScript	1362
34.2.11	developerscripttools.ReplaceInFilesScript	1365
34.2.12	developerscripttools.RunDoctestsScript	1368
34.2.13	developerscripttools.TestAndRebuildScript	1370
34.3	Functions	1372
34.3.1	developerscripttools.get_developer_script_classes	1372
34.3.2	developerscripttools.run_abjadbook	1372
34.3.3	developerscripttools.run_ajv	1372
35	documentationtools	1373
35.1	Abstract classes	1373
35.1.1	documentationtools.GraphvizObject	1373
35.1.2	documentationtools.ReSTDirective	1375
35.2	Concrete classes	1384
35.2.1	documentationtools.AbjadAPIGenerator	1384
35.2.2	documentationtools.ClassCrawler	1385
35.2.3	documentationtools.ClassDocumenter	1387
35.2.4	documentationtools.Documenter	1390
35.2.5	documentationtools.FunctionCrawler	1392
35.2.6	documentationtools.FunctionDocumenter	1393
35.2.7	documentationtools.GraphvizEdge	1395
35.2.8	documentationtools.GraphvizGraph	1397
35.2.9	documentationtools.GraphvizNode	1409
35.2.10	documentationtools.GraphvizSubgraph	1413
35.2.11	documentationtools.InheritanceGraph	1422
35.2.12	documentationtools.ModuleCrawler	1425
35.2.13	documentationtools.Pipe	1426
35.2.14	documentationtools.ReSTAutodocDirective	1428
35.2.15	documentationtools.ReSTAutosummaryDirective	1438
35.2.16	documentationtools.ReSTAutosummaryItem	1447
35.2.17	documentationtools.ReSTDDocument	1451
35.2.18	documentationtools.ReSTHeading	1461
35.2.19	documentationtools.ReSTHorizontalRule	1465
35.2.20	documentationtools.ReSTInheritanceDiagram	1469
35.2.21	documentationtools.ReSTLineageDirective	1478
35.2.22	documentationtools.ReSTOnlyDirective	1488
35.2.23	documentationtools.ReSTParagraph	1497
35.2.24	documentationtools.ReSTTOCDirective	1501
35.2.25	documentationtools.ReSTTOCItem	1510
35.2.26	documentationtools.ToolsPackageDocumenter	1514
35.3	Functions	1516
35.3.1	documentationtools.compare_images	1516
35.3.2	documentationtools.list_all_abjad_classes	1516
35.3.3	documentationtools.list_all_abjad_functions	1516
35.3.4	documentationtools.make_ligeti_example_lilypond_file	1516
35.3.5	documentationtools.make_reference_manual_graphviz_graph	1516
35.3.6	documentationtools.make_reference_manual_lilypond_file	1516
35.3.7	documentationtools.make_text_alignment_example_lilypond_file	1517
36	exceptiontools	1519
36.1	Concrete classes	1519

36.1.1	exceptiontools.AssignabilityError	1519
36.1.2	exceptiontools.ExtraSpannerError	1520
36.1.3	exceptiontools.ImpreciseTempoError	1521
36.1.4	exceptiontools.LilyPondParserError	1522
36.1.5	exceptiontools.MissingMeasureError	1523
36.1.6	exceptiontools.MissingSpannerError	1524
36.1.7	exceptiontools.MissingTempoError	1525
36.1.8	exceptiontools.OverfullContainerError	1526
36.1.9	exceptiontools.PartitionError	1527
36.1.10	exceptiontools.SchemeParserFinishedError	1528
36.1.11	exceptiontools.UnboundedTimeIntervalError	1529
36.1.12	exceptiontools.UnderfullContainerError	1530
37	lilypondnametools	1533
37.1	Concrete classes	1533
37.1.1	lilypondnametools.LilyPondGrobNameManager	1533
37.1.2	lilypondnametools.LilyPondNameManager	1534
37.1.3	lilypondnametools.LilyPondSettingNameManager	1535
38	lilypondparsertools	1537
38.1	Abstract classes	1537
38.1.1	lilypondparsertools.Music	1537
38.1.2	lilypondparsertools.SimultaneousMusic	1539
38.2	Concrete classes	1540
38.2.1	lilypondparsertools.ContextSpeccedMusic	1540
38.2.2	lilypondparsertools.GuileProxy	1541
38.2.3	lilypondparsertools.LilyPondDuration	1543
38.2.4	lilypondparsertools.LilyPondEvent	1544
38.2.5	lilypondparsertools.LilyPondFraction	1545
38.2.6	lilypondparsertools.LilyPondGrammarGenerator	1546
38.2.7	lilypondparsertools.LilyPondLexicalDefinition	1547
38.2.8	lilypondparsertools.LilyPondParser	1550
38.2.9	lilypondparsertools.LilyPondSyntacticalDefinition	1560
38.2.10	lilypondparsertools.ReducedLyParser	1577
38.2.11	lilypondparsertools.SchemeParser	1583
38.2.12	lilypondparsertools.SequentialMusic	1587
38.2.13	lilypondparsertools.SyntaxNode	1588
38.3	Functions	1589
38.3.1	lilypondparsertools.parse_reduced_ly_syntax	1589
39	systemtools	1591
39.1	Abstract classes	1591
39.1.1	systemtools.Configuration	1591
39.2	Concrete classes	1593
39.2.1	systemtools.AbjadConfiguration	1593
39.2.2	systemtools.BenchmarkScoreMaker	1597
39.2.3	systemtools.IOManager	1600
39.2.4	systemtools.ImportManager	1604
39.2.5	systemtools.LilyPondFormatBundle	1605
39.2.6	systemtools.LilyPondFormatManager	1607
39.2.7	systemtools.RedirectedStreams	1609
39.2.8	systemtools.StorageFormatManager	1610
39.2.9	systemtools.StorageFormatSpecification	1612
39.2.10	systemtools.TestManager	1614
39.2.11	systemtools.Timer	1615
39.2.12	systemtools.UpdateManager	1617
39.2.13	systemtools.WellformednessManager	1618
39.3	Functions	1620
39.3.1	systemtools.requires	1620

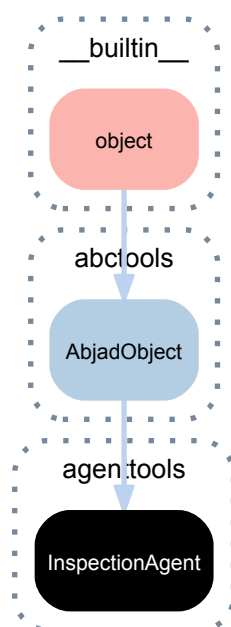
39.3.2	systemtools.run_abjad	1620
Index		1621

Part I

Core composition packages

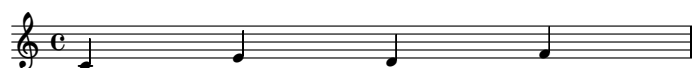
1.1 Concrete classes

1.1.1 agenttools.InspectionAgent



class `agenttools.InspectionAgent` (*client=None*)
A wrapper around the Abjad inspection methods.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



```
>>> inspect(staff)
InspectionAgent(client={c'4, e'4, d'4, f'4})
```

Bases

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

Read-only properties

`InspectionAgent.client`

Client of inspection agent.

Returns component.

Methods

`InspectionAgent.get_annotation(name, default=None)`

Gets value of annotation with *name* attached to client.

Returns *default* when no annotation with *name* is attached to client.

Raises exception when more than one annotation with *name* is attached to client.

`InspectionAgent.get_badly_formed_components()`

Gets badly formed components in client.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff[1].written_duration = Duration(1, 4)
>>> beam = spannertools.Beam()
>>> attach(beam, staff[:])
```

```
>>> inspect(staff).get_badly_formed_components()
[Note("d'4")]
```

(Beamed quarter notes are not well formed.)

Returns list.

`InspectionAgent.get_components(prototype=None, include_self=True)`

Gets all components of *prototype* in the descendants of client.

Returns client selection.

`InspectionAgent.get_contents(include_self=True)`

Gets contents of client.

Returns sequential selection.

`InspectionAgent.get_descendants(include_self=True)`

Gets descendants of client.

Returns descendants.

`InspectionAgent.get_duration(in_seconds=False)`

Gets duration of client.

Returns duration.

`InspectionAgent.get_effective(prototype=None)`

Gets effective indicator that matches *prototype* and governs client.

Returns indicator or none.

`InspectionAgent.get_effective_staff()`

Gets effective staff of client.

Returns staff or none.

`InspectionAgent.get_grace_container(kind=None)`

Gets exactly one grace container of *kind* attached to client.

Raises error when no grace container of *kind* attaches to client.

Raises error when more than one grace container of *kind* attaches to client.

Returns grace container.

`InspectionAgent.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 = scoretools.GraceContainer(
...     [Note("cs'16")],
...     kind='grace',
...     )
>>> attach(grace_container, staff[1])
>>> after_grace = scoretools.GraceContainer(
...     [Note("ds'16")],
...     kind='after'
...     )
>>> attach(after_grace, staff[1])
>>> 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.

`InspectionAgent.get_indicator` (*prototype=None, unwrap=True*)

Gets exactly one indicator matching *prototype* attached to client.

Raises exception when no indicator matching *prototype* is attached to client.

Returns indicator.

`InspectionAgent.get_indicators` (*prototype=None, unwrap=True*)

Get all indicators matching *prototype* attached to client.

Returns tuple.

`InspectionAgent.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.

`InspectionAgent.get_lineage()`
Gets lineage of client.

Returns lineage.

`InspectionAgent.get_logical_tie()`
Gets logical tie that governs leaf.

Returns logical tie.

`InspectionAgent.get_markup(direction=None)`
Gets all markup attached to client.

Returns tuple.

`InspectionAgent.get_parentage(include_self=True)`
Gets parentage of client.

Returns parentage.

`InspectionAgent.get_sounding_pitch()`
Gets sounding pitch of client.

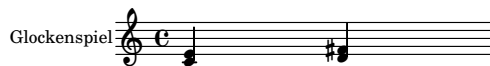
```
>>> staff = Staff("d''8 e''8 f''8 g''8")
>>> piccolo = instrumenttools.Piccolo()
>>> attach(piccolo, staff)
>>> instrumenttools.transpose_from_sounding_pitch_to_written_pitch(
...     staff)
>>> show(staff)
```



Returns named pitch.

`InspectionAgent.get_sounding_pitches()`
Gets sounding pitches of client.

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



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

Returns tuple.

`InspectionAgent.get_spanner(prototype=None)`
Gets exactly one spanner of *prototype* attached to client.

Raises exception when no spanner of *prototype* is attached to client.

Returns spanner.

`InspectionAgent.get_spanners(prototype=None)`
Gets spanners attached to client.

Returns set.

`InspectionAgent.get_timespan (in_seconds=False)`

Gets timespan of client.

Returns timespan.

`InspectionAgent.get_vertical_moment (governor=None)`

Gets vertical moment starting with client.

Returns vertical moment.

`InspectionAgent.get_vertical_moment_at (offset)`

Gets vertical moment at *offset*.

Returns vertical moment.

`InspectionAgent.has_effective_indicator (prototype=None)`

True when indicator that matches *prototype* is in effect for client. Otherwise false.

Returns boolean.

`InspectionAgent.has_indicator (prototype=None)`

True when client has one or more indicators that match *prototype*. Otherwise false.

Returns boolean.

`InspectionAgent.is_bar_line_crossing ()`

True when client crosses bar line. Otherwise false.

```
>>> staff = Staff("c'4 d'4 e'4")
>>> time_signature = TimeSignature((3, 8))
>>> attach(time_signature, staff)
>>> 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.

`InspectionAgent.is_well_formed (allow_empty_containers=True)`

True when client is well-formed. Otherwise false.

Returns false.

`InspectionAgent.report_modifications ()`

Reports modifications of client.

Report modifications of container in selection:

```
>>> container = Container("c'8 d'8 e'8 f'8")
>>> override(container).note_head.color = 'red'
>>> override(container).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.

`InspectionAgent.tabulate_well_formedness_violations()`

Tabulates well-formedness violations in client.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff[1].written_duration = Duration(1, 4)
>>> beam = spannertools.Beam()
>>> attach(beam, staff[:])

```

```

>>> result = inspect(staff).tabulate_well_formedness_violations()

```

```

>>> print result
1 / 4 beamed quarter notes
0 / 1 discontinuous spanners
0 / 5 duplicate ids
0 / 0 intermarked hairpins
0 / 0 misdurated measures
0 / 0 misfilled measures
0 / 4 mispitched ties
0 / 4 misrepresented flags
0 / 5 missing parents
0 / 0 nested measures
0 / 1 overlapping beams
0 / 0 overlapping glissandi
0 / 0 overlapping octavation spanners
0 / 0 short hairpins

```

Beamed quarter notes are not well formed.

Returns string.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

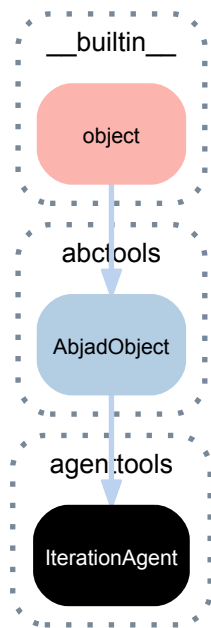
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

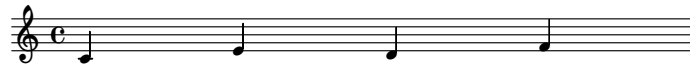
Returns string.

1.1.2 agenttools.IterationAgent



class agenttools.**IterationAgent** (*client=None*)
 A wrapper around the Abjad iteration methods.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



```
>>> iterate(staff[2:])
IterationAgent(client=SliceSelection(Note("d'4"), Note("f'4")))
```

Bases

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

Read-only properties

IterationAgent.client
 Client of iteration agent.
 Returns selection.

Methods

IterationAgent.by_class (*prototype=None, reverse=False, start=0, stop=None*)
 Iterate components forward 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 iterate(staff).by_class(Note):
...     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 iterate(staff).by_class(
...     Note, start=0, stop=3):
...     note
...
Note("c'8")
Note("d'8")
Note("e'8")
```

```
>>> for note in iterate(staff).by_class(
...     Note, 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 iterate(staff).by_class(
...     Note, 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 iterate(staff).by_class(
...     Note, reverse=True, start=3):
...     note
...
Note("e'8")
Note("d'8")
Note("c'8")
```

```
>>> for note in iterate(staff).by_class(
...     Note, reverse=True, start=0, stop=3):
...     note
...
Note("a'8")
Note("g'8")
Note("f'8")
```

```
>>> for note in iterate(staff).by_class(
...     Note, reverse=True, start=2, stop=4):
...     note
...
Note("f'8")
Note("e'8")
```

Iterates across different logical voices.

Returns generator.

`IterationAgent.by_components_and_grace_containers` (*prototype=None*)

Iterate components of *component_class* forward in *expr*:

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

>>> grace_notes = [Note("c'16"), Note("d'16")]
>>> grace = scoretools.GraceContainer(
...     grace_notes,
...     kind='grace',
... )
>>> attach(grace, voice[1])

>>> after_grace_notes = [Note("e'16"), Note("f'16")]
>>> after_grace = scoretools.GraceContainer(
...     after_grace_notes,
...     kind='after')
>>> attach(after_grace, voice[1])
```

```
>>> x = iterate(voice).by_components_and_grace_containers(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.

`IterationAgent.by_leaf_pair()`

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))
>>> clef = Clef('bass')
>>> attach(clef, score[1])
>>> show(score)
```



```
>>> for pair in iterate(score).by_leaf_pair():
...     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.

IterationAgent.**by_logical_tie** (*nontrivial=False, pitched=False, reverse=False*)

Iterate logical ties forward in *expr*:

```
>>> staff = Staff(r"c'4 ~ \times 2/3 { c'16 d'8 } e'8 f'4 ~ f'16")
```

```
>>> for x in iterate(staff).by_logical_tie():
...     x
...
LogicalTie(Note("c'4"), Note("c'16"))
LogicalTie(Note("d'8"),)
LogicalTie(Note("e'8"),)
LogicalTie(Note("f'4"), Note("f'16"))
```

Iterate logical ties backward in *expr*:

```
>>> for x in iterate(staff).by_logical_tie(reverse=True):
...     x
...
LogicalTie(Note("f'4"), Note("f'16"))
LogicalTie(Note("e'8"),)
LogicalTie(Note("d'8"),)
LogicalTie(Note("c'4"), Note("c'16"))
```

Iterate pitched logical ties in *expr*:

```
>>> for x in iterate(staff).by_logical_tie(pitched=True):
...     x
...
LogicalTie(Note("c'4"), Note("c'16"))
LogicalTie(Note("d'8"),)
LogicalTie(Note("e'8"),)
LogicalTie(Note("f'4"), Note("f'16"))
```

Iterate nontrivial logical ties in *expr*:

```
>>> for x in iterate(staff).by_logical_tie(nontrivial=True):
...     x
...
LogicalTie(Note("c'4"), Note("c'16"))
LogicalTie(Note("f'4"), Note("f'16"))
```

Returns generator.

IterationAgent.**by_logical_voice** (*component_class, logical_voice, reverse=False*)

Yield left-to-right instances of *component_class* in *expr* with *logical_voice*:

```
>>> 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
```

```
>>> for x in iterate(staff).by_logical_voice(Note, signature):
...     x
...
Note("c'8")
Note("d'8")
Note("g'8")
Note("a'8")
```

Returns generator.

`IterationAgent.by_logical_voice_from_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:

```
>>> leaf = staff.select_leaves(allow_discontiguous_leaves=True)[0]
>>> for x in iterate(leaf).by_logical_voice_from_component(Note):
...     x
...
Note("c'8")
Note("d'8")
Note("g'8")
Note("a'8")
```

Starting from the second leaf in score:

```
>>> leaf = staff.select_leaves(allow_discontiguous_leaves=True)[1]
>>> for x in iterate(leaf).by_logical_voice_from_component(Note):
...     x
...
Note("d'8")
Note("g'8")
Note("a'8")
```

Yield all components in logical voice:

```
>>> leaf = staff.select_leaves(allow_discontiguous_leaves=True)[0]
>>> for x in iterate(leaf).by_logical_voice_from_component():
...     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:

```
>>> leaf = staff.select_leaves(allow_discontiguous_leaves=True)[-1]
>>> for x in iterate(leaf).by_logical_voice_from_component(
...     Note,
...     reverse=True,
... ):
...     x
```

```
Note("c'8")
Note("b'8")
Note("f'8")
Note("e'8")
```

Yield all components in logical voice:

```
>>> leaf = staff.select_leaves(allow_discontiguous_leaves=True)[-1]
>>> for x in iterate(leaf).by_logical_voice_from_component(
...     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.

`IterationAgent.by_run` (*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 iterate(staff[:]).by_run((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 = iterate(staff).by_class(scoretools.Leaf)
```

```
>>> for group in iterate(leaves).by_run((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.

`IterationAgent.by_semantic_voice` (*reverse=False, start=0, stop=None*)

Iterate semantic voices forward in *expr*:

```
>>> measures = scoretools.make_spacer_skip_measures(
...     [(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 iterate(staff).by_semantic_voice(reverse=True):
...     voice
...
Voice="MusicVoice"{5}
```

Returns generator.

`IterationAgent.by_timeline` (*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 iterate(score).by_timeline():
...     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 iterate(score).by_timeline(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.

`IterationAgent.by_timeline_from_component` (*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 iterate(score[1][2]).by_timeline_from_component():
...     leaf
...
Note("b'8")
Note("c''8")
Note("e'4")
Note("f'4")
```

Iterate timeline backward from *component*:

```
::
```

```
>>> for leaf in iterate(score[1][2]).by_timeline_from_component(
...     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.

`IterationAgent.by_topmost_logical_ties_and_components()`

Iterate topmost logical ties and components forward in *expr*:

```
>>> string = r"c'8 ~ c'32 d'8 ~ d'32 \times 2/3 { e'8 f'8 g'8 } "  
>>> string += "a'8 ~ a'32 b'8 ~ b'32"  
>>> staff = Staff(string)
```

```
>>> for x in iterate(staff).by_topmost_logical_ties_and_components():  
...     x  
...  
LogicalTie(Note("c'8"), Note("c'32"))  
LogicalTie(Note("d'8"), Note("d'32"))  
Tuplet(Multiplier(2, 3), "e'8 f'8 g'8")  
LogicalTie(Note("a'8"), Note("a'32"))  
LogicalTie(Note("b'8"), Note("b'32"))
```

Raise logical tie error on overlapping logical ties.

Returns generator.

`IterationAgent.by_vertical_moment(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 iterate(score).by_vertical_moment():  
...     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 iterate(piano_staff).by_vertical_moment():  
...     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 iterate(score).by_vertical_moment(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 iterate(piano_staff).by_vertical_moment(
...     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.

`IterationAgent.depth_first` (*capped=True, direction=Left, forbid=None, unique=True*)
 Iterate components depth-first from *component*.

Todo

Add usage examples.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

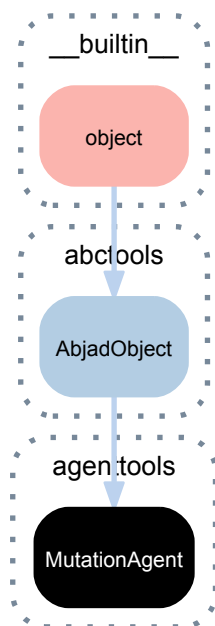
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

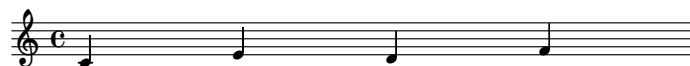
Returns string.

1.1.3 agenttools.MutationAgent



class `agenttools.MutationAgent` (*client=None*)
 A wrapper around the Abjad mutation methods.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



```
>>> mutate(staff[2:])
MutationAgent(client=SliceSelection(Note("d'4"), Note("f'4")))
```

Bases

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

Read-only properties

`MutationAgent.client`
 Returns client of mutation agent.
 Returns selection or component.

Methods

`MutationAgent.copy` (*n=1, include_enclosing_containers=False*)
 Copies component and fractures crossing spanners.
 Returns new component.

`MutationAgent.extract` (*scale_contents=False*)
 Extracts mutation client from score.
 Leaves children of mutation client in score.

Example 1. Extract tuplet:

```
>>> staff = Staff()
>>> time_signature = TimeSignature((3, 4))
>>> attach(time_signature, staff)
>>> staff.append(Tuplet((3, 2), "c'4 e'4"))
>>> staff.append(Tuplet((3, 2), "d'4 f'4"))
>>> hairpin = spannertools.Hairpin([], 'p < f')
>>> attach(hairpin, 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 = TimeSignature((3, 4))
>>> attach(time_signature, staff)
>>> staff.append(Tuplet((3, 2), "c'4 e'4"))
>>> staff.append(Tuplet((3, 2), "d'4 f'4"))
>>> hairpin = spannertools.Hairpin([], 'p < f')
>>> attach(hairpin, 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.

`MutationAgent.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 = scoretools.FixedDurationTuplet(
...     Duration(2, 8), [])
>>> tuplet_1.extend("c'8 d'8 e'8")
>>> beam = spannertools.Beam()
>>> attach(beam, tuplet_1[:])
>>> duration = Duration(2, 16)
>>> tuplet_2 = scoretools.FixedDurationTuplet(duration, [])
>>> tuplet_2.extend("c'16 d'16 e'16")
>>> slur = spannertools.Slur()
>>> attach(slur, tuplet_2[:])
>>> staff = Staff([tuplet_1, tuplet_2])
>>> show(staff)
```



```
>>> tuplets = staff[:]
>>> mutate(tuplets).fuse()
FixedDurationTuplet(Duration(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.Slur()
>>> attach(slur, 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.

`MutationAgent.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.Hairpin([], 'p < f')
>>> attach(hairpin, staff[:])
>>> slur = spannertools.Slur()
>>> attach(slur, staff.select_leaves())
>>> show(staff)
```



```
>>> notes = scoretools.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.

`MutationAgent.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.

`MutationAgent.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.

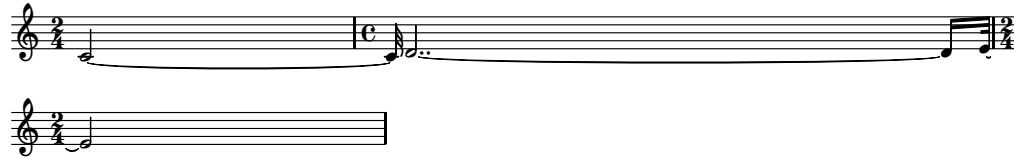
`MutationAgent.rewrite_meter(meter, boundary_depth=None, maximum_dot_count=None)`

Rewrite the contents of logical ties in an expression to match a meter.

Example 1. Rewrite the contents of a measure in a staff using the default meter 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)
```

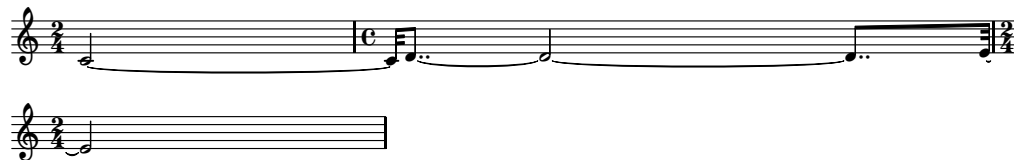
```
>>> show(staff)
```



```
>>> meter = metertools.Meter((4, 4))
>>> print meter.pretty_rtm_format
(4/4 (
  1/4
  1/4
  1/4
  1/4))
```

```
>>> mutate(staff[1][:]).rewrite_meter(meter)
```

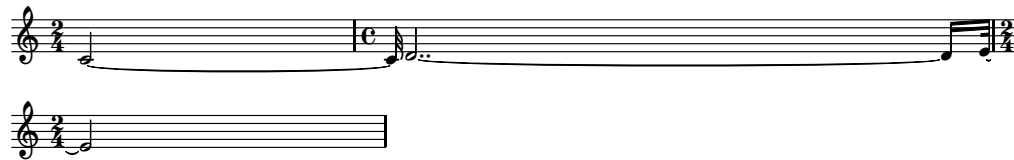
```
>>> show(staff)
```



Example 2. Rewrite the contents of a measure in a staff using a custom meter:

```
>>> staff = Staff(parseable)
```

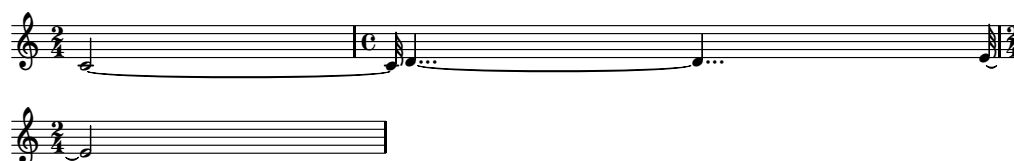
```
>>> show(staff)
```



```
>>> rtm = '(4/4 ((2/4 (1/4 1/4)) (2/4 (1/4 1/4))))'
>>> meter = metertools.Meter(rtm)
>>> print meter.pretty_rtm_format
(4/4 (
  (2/4 (
    1/4
    1/4))
  (2/4 (
    1/4
    1/4))))
```

```
>>> mutate(staff[1][:]).rewrite_meter(meter)
```

```
>>> 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 = parse(parseable)
```

```
>>> show(measure)
```



Without constraining the *maximum_dot_count*:

```
>>> mutate(measure[:]).rewrite_meter(measure)
```

```
>>> show(measure)
```



Constraining the *maximum_dot_count* to 2:

```
>>> measure = parse(parseable)
>>> mutate(measure[:]).rewrite_meter(
...     measure,
...     maximum_dot_count=2,
... )
```

```
>>> show(measure)
```



Constraining the *maximum_dot_count* to 1:

```
>>> measure = parse(parseable)
>>> mutate(measure[:]).rewrite_meter(
...     measure,
...     maximum_dot_count=1,
... )
```

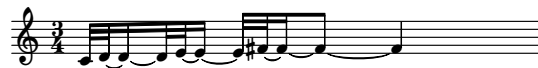
```
>>> show(measure)
```



Constraining the *maximum_dot_count* to 0:

```
>>> measure = parse(parseable)
>>> mutate(measure[:]).rewrite_meter(
...     measure,
...     maximum_dot_count=0,
... )
```

```
>>> show(measure)
```



Example 4. Split logical ties at different depths of the *Meter*, if those logical ties cross any offsets at that depth, but do not also both begin and end at any of those offsets.

Consider the default meter for 9/8:

```
>>> meter = metertools.Meter((9, 8))
>>> print meter.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 meter without specifying a *boundary_depth*:

```

>>> parseable = "abj: | 9/8 c'2 d'2 e'8 |"
>>> measure = parse(parseable)

```

```

>>> show(measure)

```



```

>>> mutate(measure[:]).rewrite_meter(measure)

```

```

>>> show(measure)

```



With a *boundary_depth* of 1, logical ties which cross any offsets created by nodes with a depth of 1 in this Meter'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 = parse(parseable)
>>> mutate(measure[:]).rewrite_meter(
...     measure,
...     boundary_depth=1,
... )

```

```

>>> show(measure)

```



For this 9/8 meter, and this input notation, A *boundary_depth* of 2 causes no change, as all logical ties already align to multiples of 1/8:

```

>>> measure = parse(parseable)
>>> mutate(measure[:]).rewrite_meter(
...     measure,
...     boundary_depth=2,
... )

```

```

>>> show(measure)

```



Example 5. Comparison of 3/4 and 6/8, at *boundary_depths* of 0 and 1:

```

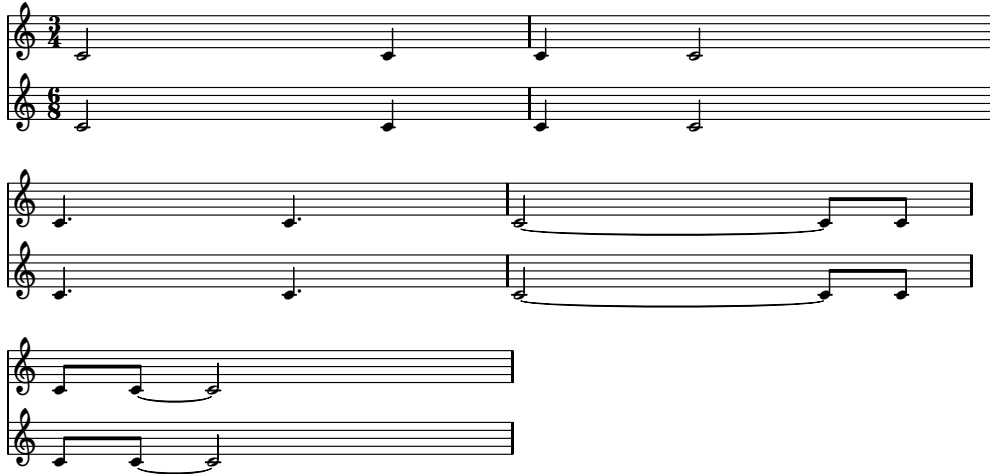
>>> triple = "abj: | 3/4 2 4 || 3/4 4 2 || 3/4 4. 4. |"
>>> triple += "| 3/4 2 ~ 8 8 || 3/4 8 8 ~ 2 |"
>>> duples = "abj: | 6/8 2 4 || 6/8 4 2 || 6/8 4. 4. |"
>>> duples += "| 6/8 2 ~ 8 8 || 6/8 8 8 ~ 2 |"
>>> score = Score([Staff(triple), Staff(duples)])

```

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 meter without specifying and boundary depth:

```
>>> for measure in iterate(score).by_class(scoretools.Measure):
...     mutate(measure[:]).rewrite_meter(measure)
```

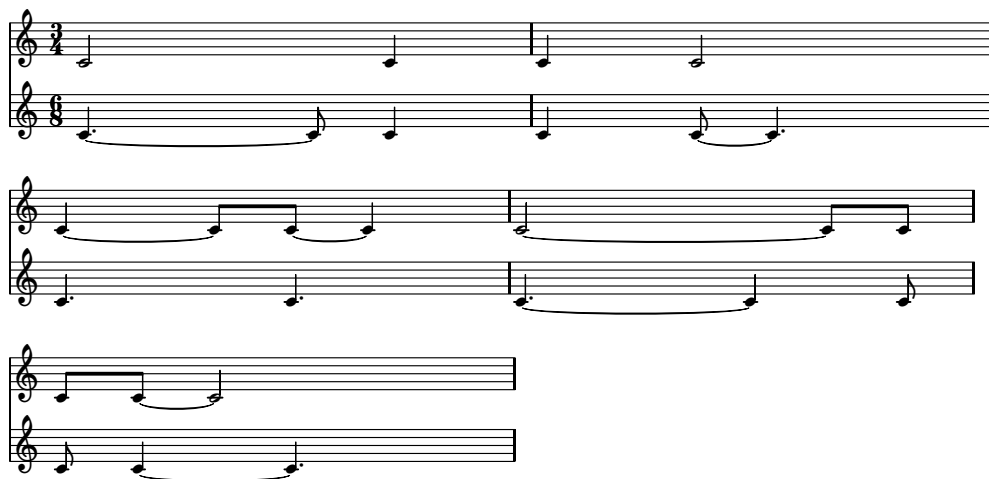
```
>>> show(score)
```



Here we re-establish meter at a boundary depth of 1:

```
>>> for measure in iterate(score).by_class(scoretools.Measure):
...     mutate(measure[:]).rewrite_meter(
...         measure,
...         boundary_depth=1,
...     )
... 
```

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



Note that the two time signatures are much more clearly disambiguated above.

Example 6. Establishing meter 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 = parse(parseable)
```

```
>>> show(measure)
```



When establishing a meter on a selection of components which contain containers, like *Tuplets* or *Containers*, `metertools.rewrite_meter()` will recurse into those containers, treating them as measures whose time signature is derived from the preprolated `preprolated_duration` of the container's contents:

```
>>> mutate(measure[:]).rewrite_meter(  
...     measure,  
...     boundary_depth=1,  
... )
```

```
>>> show(measure)
```



Operates in place and returns none.

`MutationAgent.scale(multiplier)`

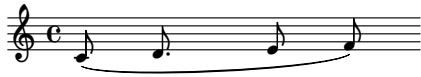
Scales mutation client by *multiplier*.

Example 1a. Scale note duration by dot-generating multiplier:

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



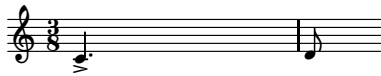
```
>>> mutate(staff[1]).scale(Multiplier(3, 2))  
>>> show(staff)
```

**Example 1b.** Scale nontrivial logical tie by dot-generating *multiplier*:

```
>>> staff = Staff(r"c'8 \accent ~ c'8 d'8")
>>> time_signature = TimeSignature((3, 8))
>>> attach(time_signature, staff)
>>> show(staff)
```



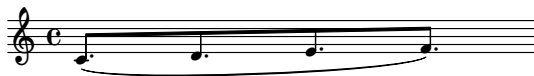
```
>>> logical_tie = inspect(staff[0]).get_logical_tie()
>>> logical_tie = mutate(logical_tie).scale(Multiplier(3, 2))
>>> show(staff)
```

**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 logical tie by tie-generating *multiplier*:

```
>>> staff = Staff(r"c'8 \accent ~ c'8 d'16")
>>> time_signature = TimeSignature((5, 16))
>>> attach(time_signature, staff)
>>> show(staff)
```



```
>>> logical_tie = inspect(staff[0]).get_logical_tie()
>>> logical_tie = mutate(logical_tie).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("c'8 ( d'8 e'8 f'8 )")
>>> show(staff)
```



```
>>> mutate(staff[1]).scale(Multiplier(2, 3))
>>> show(staff)
```

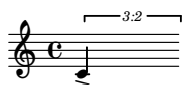


Example 3b. Scale trivial logical tie by tuplet-generating multiplier:

```
>>> staff = Staff(r"c'8 \accent")
>>> show(staff)
```



```
>>> logical_tie = inspect(staff[0]).get_logical_tie()
>>> logical_tie = mutate(logical_tie).scale(Multiplier(4, 3))
>>> show(staff)
```

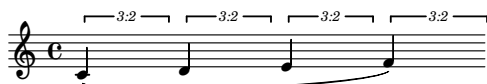


Example 3c. Scale container by tuplet-generating multiplier:

```
>>> 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")
>>> attach(Multiplier(1, 2), note)
>>> show(note)
```



```
>>> mutate(note).scale(Multiplier(5, 3))
>>> show(note)
```



Example 6. Scale tuplet:

```
>>> staff = Staff()
>>> time_signature = TimeSignature((4, 8))
>>> attach(time_signature, staff)
>>> tuplet = scoretools.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 = TimeSignature((4, 8))
>>> attach(time_signature, staff)
>>> tuplet = scoretools.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.

MutationAgent.**splice** (*components*, *direction=Right*, *grow_spanners=True*)

Splices *components* to the right or left of selection.

Returns list of components.

MutationAgent.**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()
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, leaves)
>>> override(staff).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()
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, leaves)
>>> override(staff).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()
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, leaves)
>>> override(staff).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()
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, leaves)
>>> override(staff).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()
>>> slur = spannertools.Slur()
>>> attach(slur, 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.

`MutationAgent` . **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.Hairpin([], 'p < f')
>>> attach(hairpin, leaves)
>>> measures = staff[:]
>>> slur = spannertools.Slur()
>>> attach(slur, measures)
>>> show(staff)
```



```
>>> measures = staff[:]
>>> tuplet = Tuplet(Multiplier(2, 3), [])
>>> tuplet.preferred_denominator = 4
>>> mutate(measures).swap(tuplet)
>>> show(staff)
```



Returns none.

Special methods

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

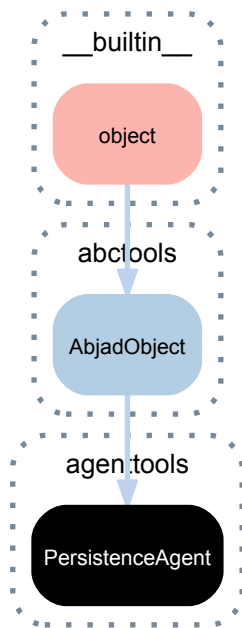
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

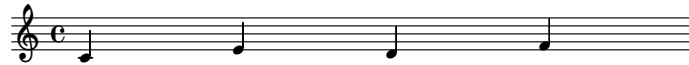
Returns string.

1.1.4 agenttools.PersistenceAgent



class `agenttools.PersistenceAgent` (*client=None*)
 A wrapper around the Abjad persistence methods.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



```
>>> persist(staff)
PersistenceAgent(client={c'4, e'4, d'4, f'4})
```

Bases

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

Read-only properties

`PersistenceAgent.client`
 Client of persistence agent.
 Returns selection or component.

Methods

`PersistenceAgent.as_ly` (*ly_filename=None*)
 Persists client as LilyPond file.
 If *ly_filename* is none, a file path will be autogenerated.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> for x in persist(staff).as_ly('~example.ly'):
...     x
...
'/Users/josiah/Desktop/test.ly'
0.04491996765136719
```

Returns output path and elapsed formatting time.

PersistenceAgent.**as_midi** (*midi_filename=None*)

Persists client as MIDI file.

If *midi_filename* is none, a file path will be autogenerated.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> for x in persist(staff).as_midi():
...     x
...
'/Users/josiah/.abjad/output/1415.midi'
0.07831692695617676
1.0882699489593506
```

Returns output path, elapsed formatting time and elapsed rendering time.

PersistenceAgent.**as_module** (*module_filename, object_name*)

Persists client as Python module.

```
>>> inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 1),
...     timespantools.Timespan(2, 4),
...     timespantools.Timespan(6, 8),
... ])
>>> persist(inventory).as_module(
...     '~/example.py', 'inventory')
```

Returns none.

PersistenceAgent.**as_pdf** (*pdf_filename=None*)

Persists client as PDF.

If *pdf_filename* is none, a file path will be autogenerated.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> for x in persist(staff).as_pdf():
...     x
...
'/Users/josiah/.abjad/output/1416.pdf'
0.047142982482910156
0.7839350700378418
```

Returns output path, elapsed formatting time and elapsed rendering time.

Special methods

(AbjadObject).**__eq__** (*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__** (*format_specification=''*)

Formats object.

Set *format_specification* to *'* or *'storage'*. Interprets *'* equal to *'storage'*.

Returns string.

(AbjadObject).**__ne__** (*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__** ()

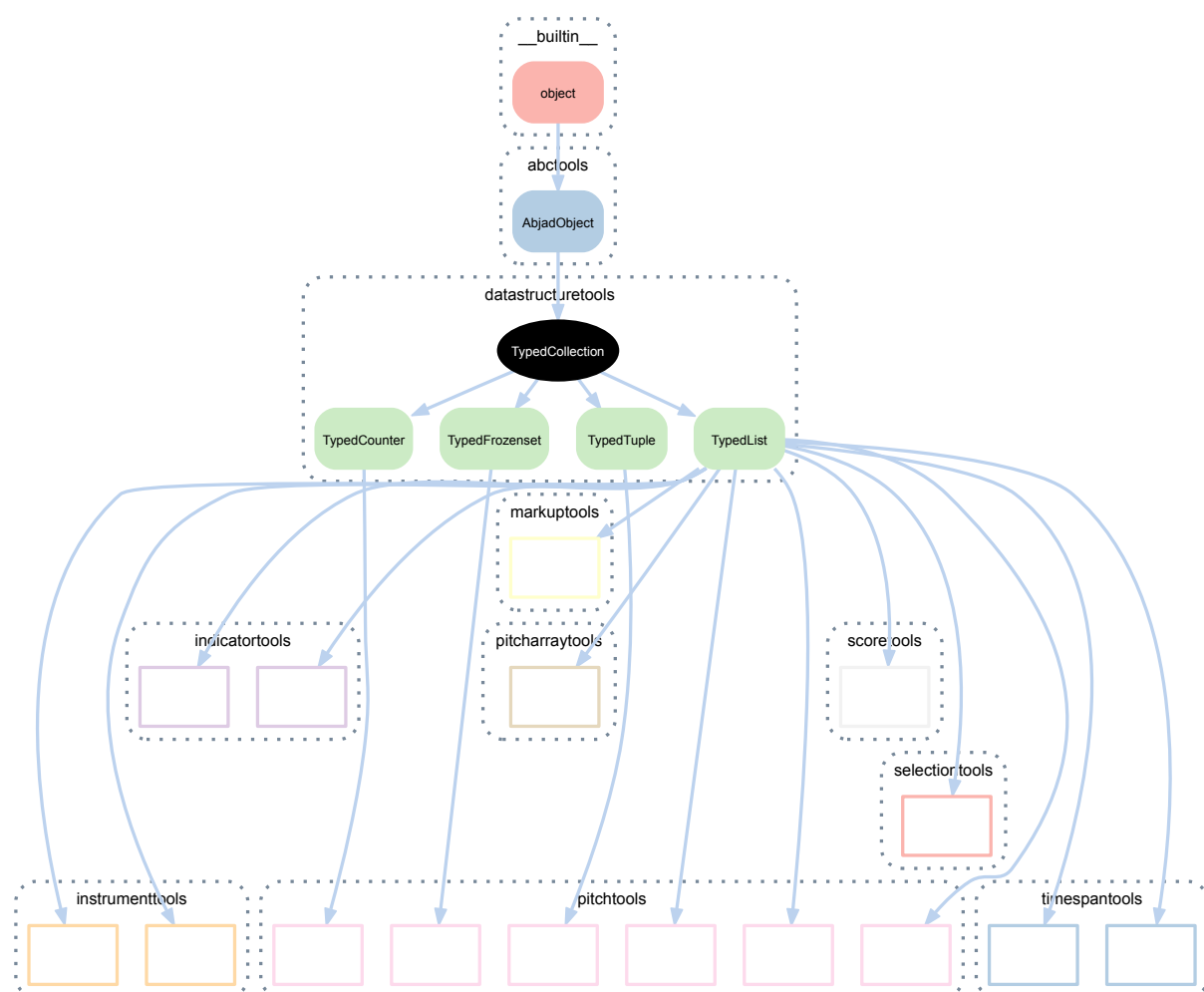
Gets interpreter representation of Abjad object.

Returns string.

DATASTRUCTURETOOLS

2.1 Abstract classes

2.1.1 datastructuretools.TypedCollection



class datastructuretools.**TypedCollection** (*tokens=None, item_class=None, custom_identifier=None*)

Abstract base class for typed collections.

Bases

- abctools.AbjadObject
- __builtin__.object

Read-only properties

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

Read/write properties

`TypedCollection.custom_identifier`
Gets and sets custom identifier of typed collection.

Returns string or none.

Special methods

`TypedCollection.__contains__ (token)`
True when typed collection container *token*. Otherwise false.

Returns boolean.

`TypedCollection.__eq__ (expr)`
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`TypedCollection.__format__ (format_specification='')`
Formats typed collection.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`TypedCollection.__iter__ ()`
Iterates typed collection.

Returns generator.

`TypedCollection.__len__ ()`
Length of typed collection.

Returns nonnegative integer.

`TypedCollection.__makenew__ (tokens=None, item_class=None, custom_identifier=None)`
Makes new typed collection with optional new values.

Returns new typed collection.

`TypedCollection.__ne__ (expr)`
True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

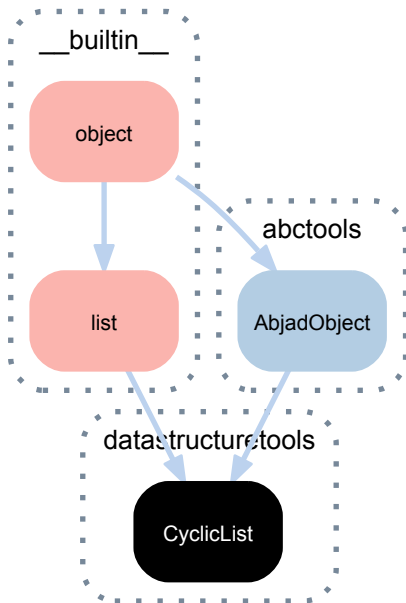
Returns boolean.

`(AbjadObject).__repr__ ()`
Gets interpreter representation of Abjad object.

Returns string.

2.2 Concrete classes

2.2.1 datastructuretools.CyclicList



class datastructuretools.**CyclicList**
A 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

- `abctools.AbjadObject`
- `__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.

CyclicList **.__eq__** (expr)
 True when *expr* has items equal to those of this cyclic list. Otherwise false.
 Returns boolean.

(AbjadObject) **.__format__** (format_specification='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(list) **.__ge__** ()
 x.__ge__(y) <==> x>=y

CyclicList **.__getitem__** (i)
 Gets *i* from cyclic list.
 Raise index error when *i* can not be found in cyclic list.
 Returns item.

CyclicList **.__getslice__** (start_index, stop_index)
 Gets items in cyclic list from *start_index* to *stop_index*.
 Returns list.

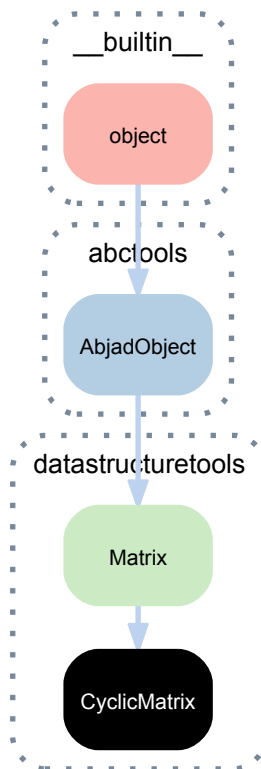
(list) **.__gt__** ()
 x.__gt__(y) <==> x>y

```

(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
(ObjadObject).__ne__(expr)
    Is true when Abjad object does not equal expr. Otherwise false.
    Returns boolean.
(ObjadObject).__repr__()
    Gets interpreter representation of Abjad object.
    Returns string.
(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__()
    String representation of cyclic list.
    Returns string.

```

2.2.2 datastructuretools.CyclicMatrix



class datastructuretools.**CyclicMatrix**(*args, **kwargs)
A cyclic matrix.

Initializes 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
```

Initializes 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
```

Only item retrieval is currently implemented.

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 of cyclic matrix.

```
>>> cyclic_matrix = datastructuretools.CyclicMatrix([
...     [0, 1, 2, 3],
...     [10, 11, 12, 13],
...     [20, 21, 22, 23],
...     ])
```

```
>>> print format(cyclic_matrix.columns)
datastructuretools.CyclicTuple(
    [
        datastructuretools.CyclicTuple(
            [0, 10, 20]
        ),
        datastructuretools.CyclicTuple(
            [1, 11, 21]
        ),
        datastructuretools.CyclicTuple(
            [2, 12, 22]
        ),
        datastructuretools.CyclicTuple(
            [3, 13, 23]
        ),
    ]
)
```

Returns cyclic tuple.

`CyclicMatrix.rows`

Rows of cyclic matrix.

```
>>> cyclic_matrix = datastructuretools.CyclicMatrix([
...     [0, 1, 2, 3],
...     [10, 11, 12, 13],
...     [20, 21, 22, 23],
...     ])
```

```
>>> print format(cyclic_matrix.rows)
datastructuretools.CyclicTuple(
  [
    datastructuretools.CyclicTuple(
      [0, 1, 2, 3]
    ),
    datastructuretools.CyclicTuple(
      [10, 11, 12, 13]
    ),
    datastructuretools.CyclicTuple(
      [20, 21, 22, 23]
    ),
  ]
)
```

Returns cyclic tuple.

Special methods

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

CyclicMatrix.**__getitem__**(*i*)

Gets row *i* from cyclic matrix.

Returns row.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

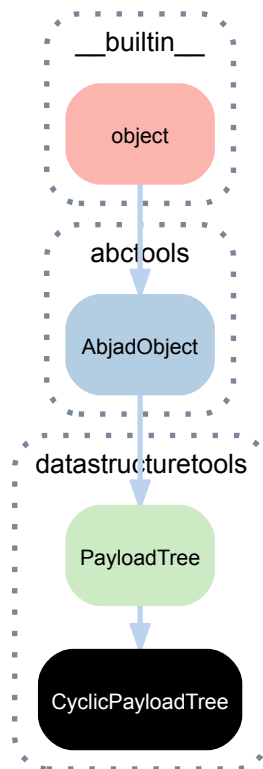
Returns boolean.

CyclicMatrix.**__repr__**()

Gets interpreter representation of cyclic matrix.

Returns string.

2.2.3 datastructuretools.CyclicPayloadTree



class datastructuretools.CyclicPayloadTree (*expr=None*)

A cyclic payload tree.

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 payload trees raise no index errors.

Here is a cyclic payload tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.CyclicPayloadTree(sequence)
```

```
>>> tree
CyclicPayloadTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Here's an internal node:

```
>>> tree[2]
CyclicPayloadTree([4, 5])
```

Here's the same node indexed with a different way:

```
>>> tree[2]
CyclicPayloadTree([4, 5])
```

With a negative index:

```
>>> tree[-2]
CyclicPayloadTree([4, 5])
```

And another negative index:

```
>>> tree[-6]
CyclicPayloadTree([4, 5])
```

Here's a leaf node:

```
>>> tree[2][0]
CyclicPayloadTree(4)
```

And here's the same node indexed a different way:

```
>>> 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 payload tree.

```
>>> 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 payload tree.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].depth
2
```

Returns nonnegative integer.

(`PayloadTree`) **.graphviz_format**

Graphviz format of payload tree.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
>>> print tree.graphviz_format
digraph G {
    node_0 [label="",
            shape=circle];
    node_1 [label="",
            shape=circle];
    node_2 [label=0,
            shape=box];
    node_3 [label=1,
            shape=box];
    node_4 [label="",
            shape=circle];
    node_5 [label=2,
            shape=box];
    node_6 [label=3,
```

```

        shape=box];
node_7 [label="",
        shape=circle];
node_8 [label=4,
        shape=box];
node_9 [label=5,
        shape=box];
node_10 [label="",
         shape=circle];
node_11 [label=6,
         shape=box];
node_12 [label=7,
         shape=box];
node_0 -> node_1;
node_0 -> node_10;
node_0 -> node_4;
node_0 -> node_7;
node_1 -> node_2;
node_1 -> node_3;
node_10 -> node_11;
node_10 -> node_12;
node_4 -> node_5;
node_4 -> node_6;
node_7 -> node_8;
node_7 -> node_9;
    }

```

Returns string.

`(PayloadTree).graphviz_graph`

The GraphvizGraph representation of payload tree.

```

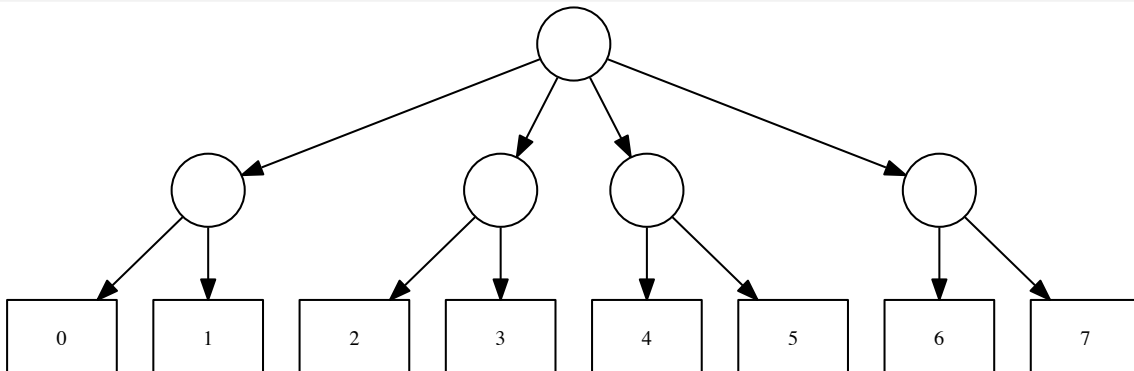
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> graph = tree.graphviz_graph
>>> topleveltools.graph(graph)

```



Returns graphviz graph.

`(PayloadTree).improper_parentage`

Improper parentage of payload tree.

```

>>> 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 payload 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 payload 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) **.width**
Number of leaves in payload tree.

```
>>> 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*)
Gets manifest payload of next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Trims first node if necessary.

Returns list of arbitrary values.

(PayloadTree).**get_next_n_complete_nodes_at_level**(*n*, *level*)

Gets next *n* complete nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Gets 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)]
```

Gets 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])]
```

Gets 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)]
```

Gets 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])]
```

Gets 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)]
```

Gets 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])]
```

Gets 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)]
```

Gets 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])]
```

Trims first node if necessary.

Returns list of nodes.

`CyclicPayloadTree.get_next_n_nodes_at_level(n, level)`

Gets next *n* nodes of cyclic payload tree at *level*.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.CyclicPayloadTree(sequence)
```

Gets next 4 nodes at level 2:

```
>>> for x in tree[0][0].get_next_n_nodes_at_level(4, 2):
...     x
CyclicPayloadTree(1)
CyclicPayloadTree(2)
CyclicPayloadTree(3)
CyclicPayloadTree(4)
```

Gets 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

Gets previous 4 nodes at level 2:

```
>>> for x in tree[0][0].get_next_n_nodes_at_level(-4, 2):
...     x
CyclicPayloadTree(7)
CyclicPayloadTree(6)
CyclicPayloadTree(5)
CyclicPayloadTree(4)
```

Gets 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)`
 Gets cyclic payload tree node at *position*.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.CyclicPayloadTree(sequence)
```

```
>>> tree.get_node_at_position((2, 1))
CyclicPayloadTree(5)
```

```
>>> tree.get_node_at_position((2, 99))
CyclicPayloadTree(5)
```

```
>>> tree.get_node_at_position((82, 1))
CyclicPayloadTree(5)
```

```
>>> tree.get_node_at_position((82, 99))
CyclicPayloadTree(5)
```

Returns node.

`(PayloadTree).get_position_of_descendant(descendant)`
 Gets 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
```

Otherwise false:

```
>>> 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)`

Iterates 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([6, 7])
PayloadTree([4, 5])
PayloadTree([2, 3])
PayloadTree([0, 1])
```

```
>>> for x in tree.iterate_at_level(1, reverse=True): x
...
PayloadTree(7)
PayloadTree(6)
PayloadTree(5)
```

```
>>> 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*)
Iterates 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 cyclic payload tree tree depth first.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.CyclicPayloadTree(sequence)
```

Example 1. Iterates from left to right:

```
>>> generator = 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. Iterates from right to left:

```
>>> generator = 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)
```

Yields cyclic payload tree nodes.

(`PayloadTree`).`iterate_payload` (*reverse=False*)
Iterates payload of tree.

Example 1. Iterates 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. Iterates 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*)
Removes *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*)
Removes 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]])
```

Modifies in-place to root.

Returns none.

(PayloadTree) **.to_nested_lists** ()
Changes 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 payload 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.

`CyclicPayloadTree.__format__(format_specification='')`
Formats cyclic tree.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(PayloadTree).__getitem__(expr)`
Gets *expr* from payload tree.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[-1]
PayloadTree([6, 7])
```

Gets slice from payload tree:

```
>>> tree[-2:]
[PayloadTree([4, 5]), PayloadTree([6, 7])]
```

Returns node.

`CyclicPayloadTree.__iter__()`
Iterates cyclic payload tree.

```
>>> for x in tree:
...     x
CyclicPayloadTree([0, 1])
CyclicPayloadTree([2, 3])
CyclicPayloadTree([4, 5])
CyclicPayloadTree([6, 7])
```

Yields cyclic payload tree nodes.

`(PayloadTree).__len__()`
Number of children in payload tree.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> len(tree)
4
```

Returns nonnegative integer.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(PayloadTree).__repr__()`
Gets interpreter representation of payload 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.

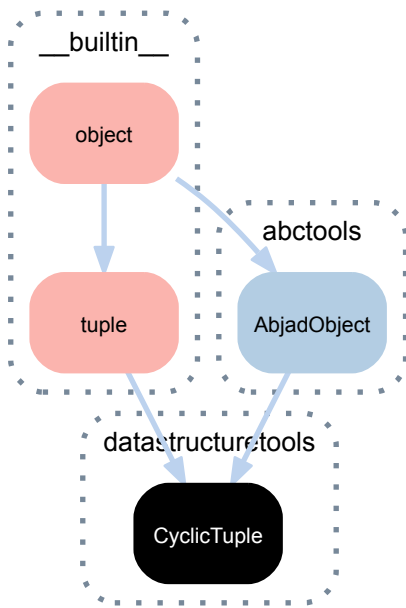
`(PayloadTree).__str__()`
String representation of payload tree.

```
>>> 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.

2.2.4 datastructuretools.CyclicTuple



class datastructuretools.CyclicTuple
A 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

- `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`

`CyclicTuple.__eq__(expr)`
 True when *expr* is a tuple with items equal to those of this cyclic tuple. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`(tuple).__ge__()`
`x.__ge__(y) <==> x>=y`

`CyclicTuple.__getitem__(i)`
 Gets *i* from cyclic tuple.
 Raises index error when *i* can not be found in cyclic tuple.
 Returns item.

`CyclicTuple.__getslice__(start_index, stop_index)`
 Gets slice of items from *start_index* to *stop_index* in cyclic tuple.
 Returns tuple.

`(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`

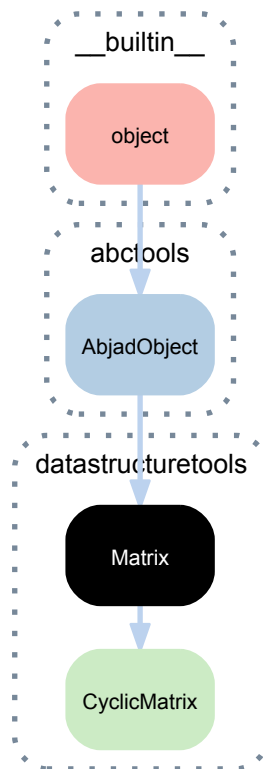
`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

`(tuple).__rmul__()`
`x.__rmul__(n) <==> n*x`

`CyclicTuple.__str__()`
 String representation of cyclic tuple.
 Returns string.

2.2.5 datastructuretools.Matrix



class datastructuretools.**Matrix**(*args, **kwargs)

A matrix.

Initializes 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
```

Initializes 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 currently implements only item retrieval.

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 of matrix.

```
>>> 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 of matrix.

```
>>> 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)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`Matrix.__getitem__(i)`

Gets row *i* from matrix.

```
>>> matrix[1]
(10, 11, 12, 13)
```

Returns row.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

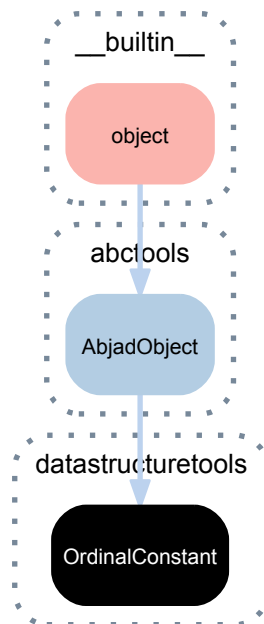
Matrix.**__repr__**()

Gets interpreter representation of matrix.

```
>>> matrix
Matrix(3x4)
```

Returns string.

2.2.6 datastructuretools.OrdinalConstant



class datastructuretools.OrdinalConstant

An ordinal constant.

Initializes 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 pytest
```

```
>>> bool(pytest.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`

Special methods

`OrdinalConstant.__eq__(expr)`

True when *expr* is an ordinal constant with dimension and value equal to those of this ordinal constant. Otherwise false.

Returns boolean.

`OrdinalConstant.__format__(format_specification='')`

Formats ordinal constant.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`OrdinalConstant.__ge__(other)`

`x.__ge__(y) <==> x>=y`

`OrdinalConstant.__gt__(other)`

`x.__gt__(y) <==> x>y`

`OrdinalConstant.__le__(other)`

`x.__le__(y) <==> x<=y`

`OrdinalConstant.__lt__(expr)`

True when *expr* is an ordinal with value greater than that of this ordinal constant. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

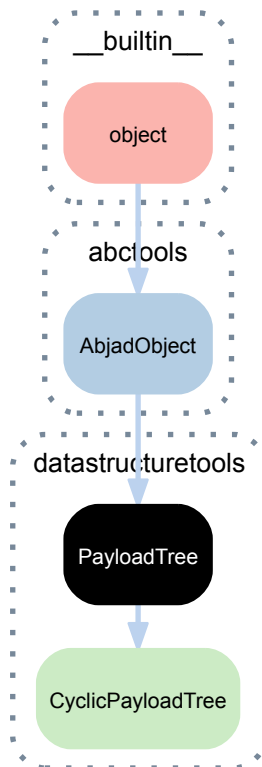
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`OrdinalConstant.__new__(dimension=None, value=0, representation=None)`

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

2.2.7 datastructuretools.PayloadTree



class `datastructuretools.PayloadTree` (*expr=None*)
 A payload tree.

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
```

```
>>> for x in tree.children:
...     x
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 payload tree.

```
>>> 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 payload tree.


```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[1].depth
2
```

Returns nonnegative integer.

`PayloadTree.graphviz_format`
Graphviz format of payload tree.

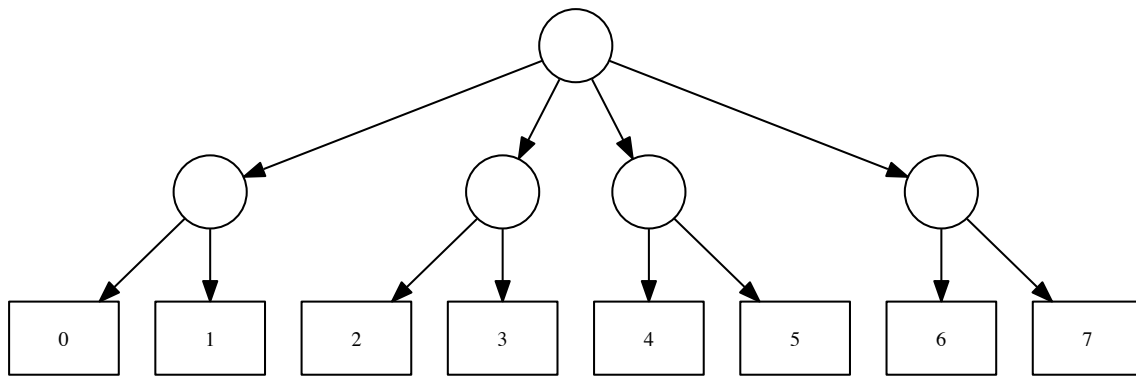
```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
>>> print tree.graphviz_format
digraph G {
    node_0 [label="",
            shape=circle];
    node_1 [label="",
            shape=circle];
    node_2 [label=0,
            shape=box];
    node_3 [label=1,
            shape=box];
    node_4 [label="",
            shape=circle];
    node_5 [label=2,
            shape=box];
    node_6 [label=3,
            shape=box];
    node_7 [label="",
            shape=circle];
    node_8 [label=4,
            shape=box];
    node_9 [label=5,
            shape=box];
    node_10 [label="",
            shape=circle];
    node_11 [label=6,
            shape=box];
    node_12 [label=7,
            shape=box];
    node_0 -> node_1;
    node_0 -> node_10;
    node_0 -> node_4;
    node_0 -> node_7;
    node_1 -> node_2;
    node_1 -> node_3;
    node_10 -> node_11;
    node_10 -> node_12;
    node_4 -> node_5;
    node_4 -> node_6;
    node_7 -> node_8;
    node_7 -> node_9;
}
```

Returns string.

`PayloadTree.graphviz_graph`
The GraphvizGraph representation of payload tree.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> graph = tree.graphviz_graph
>>> topleveltools.graph(graph)
```



Returns graphviz graph.

PayloadTree.improper_parentage

Improper parentage of payload tree.

```
>>> 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 payload 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 payload 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.width`
Number of leaves in payload tree.

```
>>> 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*)

Gets manifest payload of next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Gets 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]
```

Trims first node if necessary.

Returns list of arbitrary values.

`PayloadTree.get_next_n_complete_nodes_at_level` (*n*, *level*)

Gets next *n* complete nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Gets 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)]
```

Gets 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])]
```

Gets 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)]
```

Gets 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])]
```

Gets 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)]
```

Gets 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])]
```

Gets 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)]
```

Gets 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])]
```

Trims first node if necessary.

Returns list of nodes.

`PayloadTree.get_next_n_nodes_at_level(n, level)`

Gets next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

Gets 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)]
```

Gets 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])]
```

Gets next node at level 0:

```
>>> tree[0][0].get_next_n_nodes_at_level(1, 0)
[PayloadTree([1], [2, 3], [4, 5], [6, 7])]
```

Gets 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)]
```

Gets 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])]
```

Gets 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)]
```

Gets 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])]
```

Gets previous node at level 0:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-1, 0)
[PayloadTree([[0, 1], [2, 3], [4, 5], [6]])]
```

Gets 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)]
```

Gets 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])]
```

Trims first node if necessary.

Returns list of nodes.

`PayloadTree.get_node_at_position(position)`

Gets 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)`

Gets 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
```

Otherwise false:

```
>>> 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*)

Iterates 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*)
Iterates 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)`
Iterates payload of tree.

Example 1. Iterates 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. Iterates 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)`
Removes *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*)

Removes 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]])
```

Modifies in-place to root.

Returns none.

`PayloadTree.to_nested_lists`()

Changes 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 payload 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.__format__(format_specification='')`

Formats payload tree.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> print format(tree)
datastructuretools.PayloadTree(
[
    [0, 1],
    [2, 3],
    [4, 5],
    [6, 7],
]
)
```

Returns string.

`PayloadTree.__getitem__(expr)`

Gets *expr* from payload tree.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> tree[-1]
PayloadTree([6, 7])
```

Gets slice from payload tree:

```
>>> tree[-2:]
[PayloadTree([4, 5]), PayloadTree([6, 7])]
```

Returns node.

`PayloadTree.__len__()`

Number of children in payload tree.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
```

```
>>> len(tree)
4
```

Returns nonnegative integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

PayloadTree.**__repr__**()

Gets interpreter representation of payload 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__**()

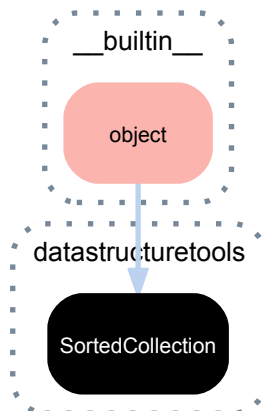
String representation of payload tree.

```
>>> 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.

2.2.8 datastructuretools.SortedCollection



class datastructuretools.**SortedCollection** (*iterable=()*, *key=None*)

A sorted collection.

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()`
Clears sorted collection.

Returns none.

`SortedCollection.copy()`

Copies sorted collection.

Returns new sorted collection.

`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)`

True when sorted collection contains *item*. Otherwise false.

Returns boolean.

`SortedCollection.__getitem__(i)`

Gets *i* in sorted collection.

Returns item.

`SortedCollection.__iter__()`

Iterates sorted collection.

Yields items.

`SortedCollection.__len__()`

Length of sorted collection.

Defined equal to number of items in collection.

Returns nonnegative integer.

`SortedCollection.__repr__()`

Interpreter representation of sorted collection.

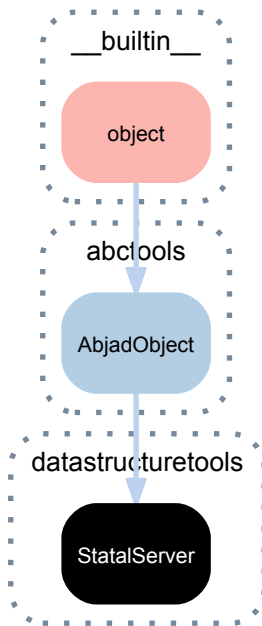
Returns string.

`SortedCollection.__reversed__()`

Reverses sorted collection.

Yields items.

2.2.9 datastructuretools.StatalServer



class `datastructuretools.StatalServer` (*cyclic_tree=None*)
A statal server.

Bases

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

Read-only properties

`StatalServer.cyclic_tree`
Statal server cyclic tree.

`StatalServer.last_node`
Statal server last node.

Special methods

`StatalServer.__call__` (*position=None, reverse=False*)
Calls statal server.

Returns statal server cursor.

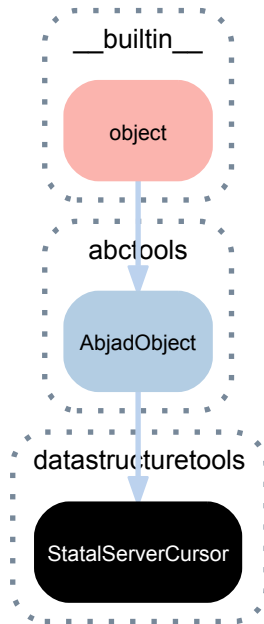
`StatalServer.__eq__` (*expr*)
True when *expr* is a statal server with cyclic tree equal to that of this statal server. Otherwise false.
Returns boolean.

`(AbjadObject).__format__` (*format_specification=''*)
Formats object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

2.2.10 datastructuretools.StatalServerCursor



class datastructuretools.**StatalServerCursor** (*statal_server=None, position=None, reverse=False*)
 A statal server cursor.

Bases

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

Read-only properties

StatalServerCursor.position
 Statal server cursor position.
 Returns tuple.

StatalServerCursor.reverse
 Statal server cursor reverse.
 False when cursor reads from left to right. True when cursor reads from right to left.
 Returns boolean.

StatalServerCursor.statal_server
 Statal server cursor statal server.
 Returns statal server.

Special methods

`StatalServerCursor.__call__(n=1, level=-1)`

Get manifest payload of next *n* nodes at *level*.

Returns list of arbitrary values.

`StatalServerCursor.__eq__(expr)`

True *expr* is a statal server cursor and keyword argument values are equal. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

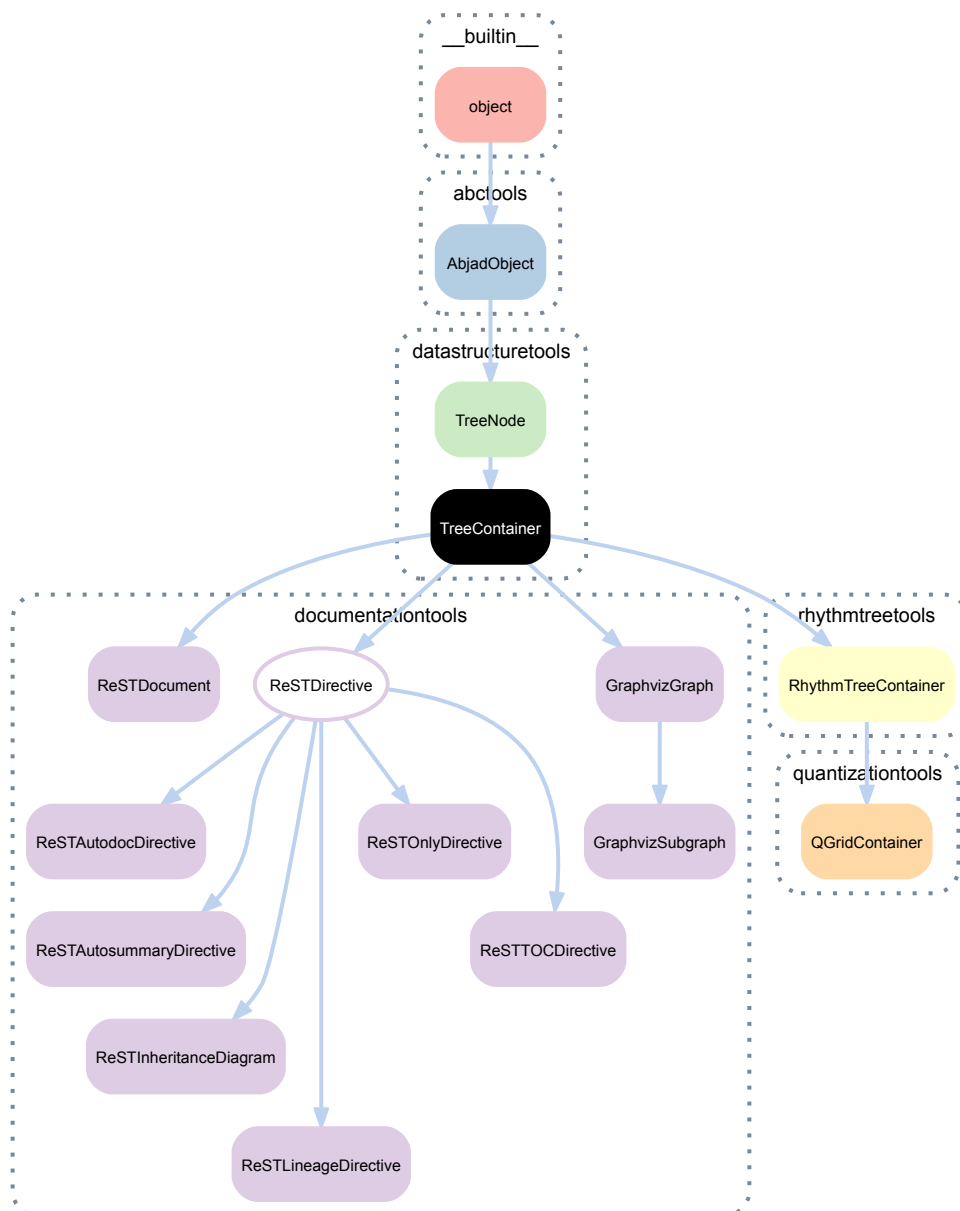
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

2.2.11 datastructuretools.TreeContainer



class datastructuretools.**TreeContainer** (*children=None, name=None*)
 A tree container.

Inner node in a generalized tree data structure.

```
>>> a = datastructuretools.TreeContainer()
>>> a
TreeContainer()
```

```
>>> b = datastructuretools.TreeNode()
>>> a.append(b)
>>> a
TreeContainer(
  children=(
    TreeNode(),
  )
)
```

Bases

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`TreeContainer.children`

Children of tree container.

```
>>> 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 tree nodes.

`(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**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

TreeContainer **.leaves**

Leaves of tree container.

```
>>> 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 tree nodes produced by iterating tree container 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**
Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(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
```

Returns tree node.

Read/write properties

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

TreeContainer **.append** (*node*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

TreeContainer **.extend** (*expr*)

Extendes *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

TreeContainer.index (*node*)

Indexes *node* in tree 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 tree 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*)

Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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)
Copies tree node.
```

Returns new tree node.

```
(TreeNode) .__deepcopy__ (*args)
Copies tree node.
```

Returns new tree node.

```
TreeContainer.__delitem__(i)
Deletes node i in tree container.
```

```
>>> 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 expr is a tree container with type, duration and children equal to this tree container. Otherwise false.
```

Returns boolean.

```
(AbjadObject) .__format__(format_specification='')
Formats object.
```

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

```
TreeContainer.__getitem__(i)
Gets node i in tree 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.

`TreeContainer.__iter__()`
Iterates tree container.

Yields children of tree container.

`TreeContainer.__len__()`
Returns nonnegative integer number of nodes in container.

`(TreeNode).__ne__(expr)`
True when tree node does not equal *expr*. Otherwise false.
Returns boolean.

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

`TreeContainer.__setitem__(i, expr)`
Sets *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
```

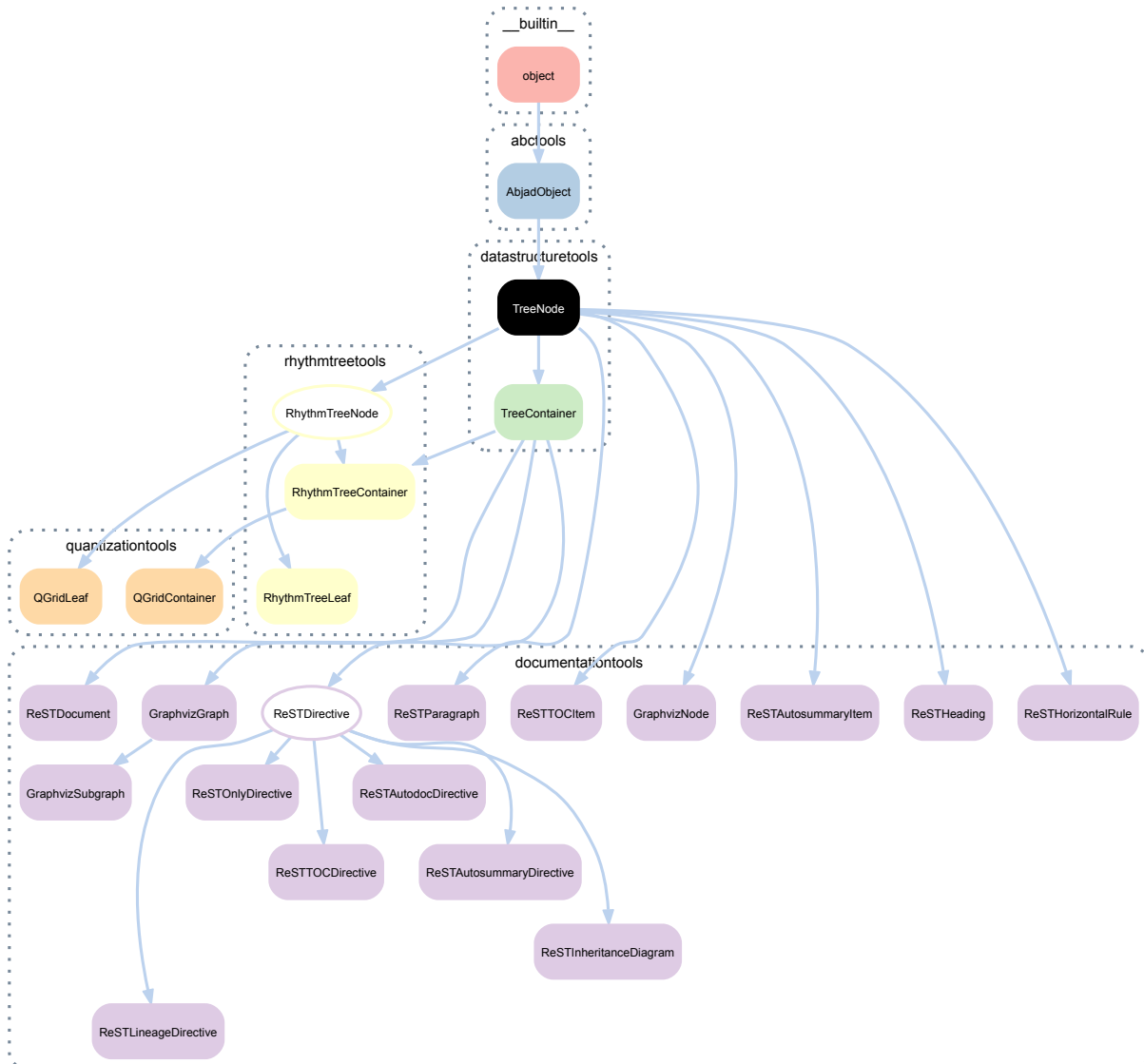
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

2.2.12 datastructuretools.TreeNode



class datastructuretools.**TreeNode** (*name=None*)

A node.

Node in a generalized tree.

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**

Graph order of tree node.

Returns tuple.

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 tree nodes.

TreeNode.**parent**

Parent of tree 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
```

Returns tree node.

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 tree nodes.

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
```

Returns tree node.

Read/write properties

`TreeNode.name`

Named of tree node.

Returns string.

Special methods

`TreeNode.__copy__(*args)`

Copies tree node.

Returns new tree node.

`TreeNode.__deepcopy__(*args)`

Copies tree node.

Returns new tree node.

`TreeNode.__eq__(expr)`

True when *expr* is a tree node. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`TreeNode.__ne__(expr)`

True when tree node does not equal *expr*. Otherwise false.

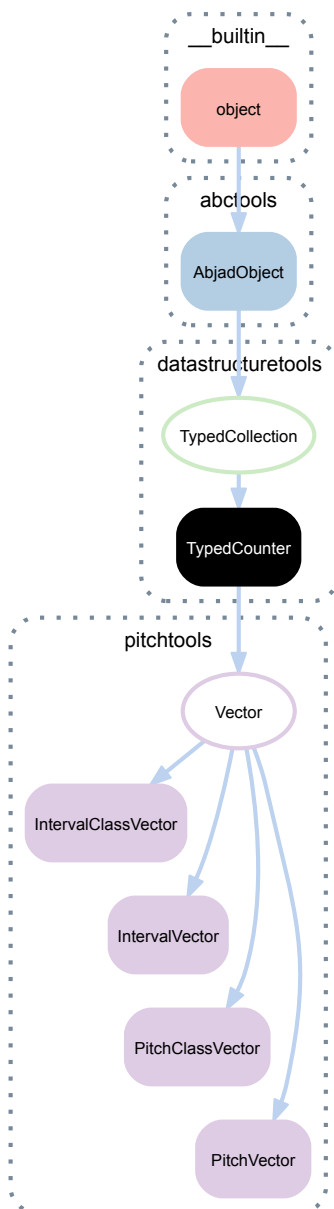
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

2.2.13 datastructuretools.TypedCounter



class datastructuretools.**TypedCounter** (*tokens=None*, *item_class=None*, *custom_identifier=None*, ***kwargs*)

A typed counter.

```
>>> counter = datastructuretools.TypedCounter(
...     [0, "c'", 1, True, "cs'", "df'"],
...     item_class=pitchtools.NumberedPitch,
... )
```

```
>>> print format(counter)
datastructuretools.TypedCounter(
{
    pitchtools.NumberedPitch(0): 2,
    pitchtools.NumberedPitch(1): 4,
},
item_class=pitchtools.NumberedPitch,
)
```

Bases

- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

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

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

`TypedCounter.clear()`
Clears typed counter.

Returns none.

`TypedCounter.copy()`
Copies typed counter.

Returns new typed counter.

`TypedCounter.elements()`
Elements in typed counter.

`TypedCounter.items()`
Items in typed counter.

Returns tuple.

`TypedCounter.iteritems()`
Iterates items in typed counter.

Yields items.

`TypedCounter.iterkeys()`
Iterates keys in typed counter.

`TypedCounter.itervalues()`
Iterates values in typed counter.

`TypedCounter.keys()`
Keys in typed counter.

`TypedCounter.most_common(n=None)`
Please document.

`TypedCounter.subtract(iterable=None, **kwargs)`
Stracts *iterable* from typed counter.

`TypedCounter.update(iterable=None, **kwargs)`
Updates typed counter with *iterable*.

`TypedCounter.values()`
Values of typed counter.

`TypedCounter.viewitems()`
Please document.

`TypedCounter.viewkeys()`
Please document.

`TypedCounter.viewvalues()`
Please document.

Special methods

`TypedCounter.__add__(expr)`
Adds typed counter to *expr*.

Returns new typed counter.

`TypedCounter.__and__(expr)`
Logical AND of typed counter and *expr*.

Returns new typed counter.

`(TypedCollection).__contains__(token)`
True when typed collection container *token*. Otherwise false.

Returns boolean.

`TypedCounter.__delitem__(token)`
Deletes *token* from typed counter.

Returns none.

`(TypedCollection).__eq__(expr)`
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`
Formats typed collection.

Set *format_specification* to ' or 'storage'. Interprets ' equal to 'storage'.

Returns string.

`TypedCounter.__getitem__(token)`
Gets *token* from typed counter.

Returns item.

`(TypedCollection).__iter__()`
Iterates typed collection.

Returns generator.

`(TypedCollection).__len__()`
Length of typed collection.

Returns nonnegative integer.

`(TypedCollection).__makenew__(tokens=None, item_class=None, custom_identifier=None)`
Makes new typed collection with optional new values.

Returns new typed collection.

`TypedCounter.__missing__(token)`
Returns zero.

Returns zero.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

TypedCounter.**__or__**(*expr*)

Logical OR of typed counter and *expr*.

Returns new typed counter.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

TypedCounter.**__setitem__**(*token*, *value*)

Sets typed counter *token* to *value*.

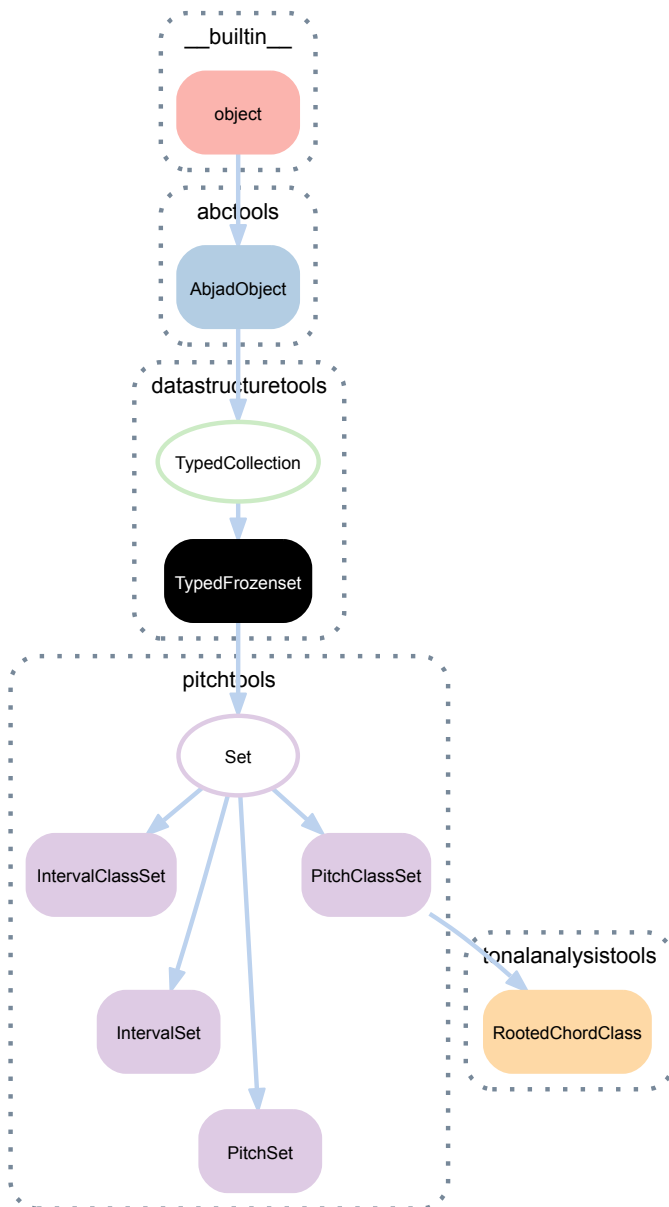
Returns none.

TypedCounter.**__sub__**(*expr*)

Subtracts *expr* from typed counter.

Returns new typed counter.

2.2.14 datastructuretools.TypedFrozenset



class datastructuretools.**TypedFrozenset** (*tokens=None*, *item_class=None*, *custom_identifier=None*)

A typed frozen set.

Bases

- datastructuretools.TypedCollection
- abctools.AbjadObject
- __builtin__.object

Read-only properties

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

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

Methods

`TypedFrozenSet.copy()`
Copies typed frozen set.

Returns new typed frozen set.

`TypedFrozenSet.difference(expr)`
Typed frozen set set-minus *expr*.

Returns new typed frozen set.

`TypedFrozenSet.intersection(expr)`
Set-theoretic intersection of typed frozen set and *expr*.

Returns new typed frozen set.

`TypedFrozenSet.isdisjoint(expr)`
True when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

`TypedFrozenSet.issubset(expr)`
True when typed frozen set is a subset of *expr*. Otherwise false.

Returns boolean.

`TypedFrozenSet.issuperset(expr)`
True when typed frozen set is a superset of *expr*. Otherwise false.

Returns boolean.

`TypedFrozenSet.symmetric_difference(expr)`
Symmetric difference of typed frozen set and *expr*.

Returns new typed frozen set.

`TypedFrozenSet.union(expr)`
Union of typed frozen set and *expr*.

Returns new typed frozen set.

Special methods

`TypedFrozenSet.__and__(expr)`
Logical AND of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedCollection).__contains__(token)`
True when typed collection container *token*. Otherwise false.

Returns boolean.

`(TypedCollection).__eq__(expr)`
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`
 Formats typed collection.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`TypedFrozenSet.__ge__(expr)`
 True when typed frozen set is greater than or equal to *expr*. Otherwise false.
 Returns boolean.

`TypedFrozenSet.__gt__(expr)`
 True when typed frozen set is greater than *expr*. Otherwise false.
 Returns boolean.

`TypedFrozenSet.__hash__()`
 Hashes typed frozen set.
 Returns integer.

`(TypedCollection).__iter__()`
 Iterates typed collection.
 Returns generator.

`TypedFrozenSet.__le__(expr)`
 True when typed frozen set is less than or equal to *expr*. Otherwise false.
 Returns boolean.

`(TypedCollection).__len__()`
 Length of typed collection.
 Returns nonnegative integer.

`TypedFrozenSet.__lt__(expr)`
 True when typed frozen set is less than *expr*. Otherwise false.
 Returns boolean.

`(TypedCollection).__makenew__(tokens=None, item_class=None, custom_identifier=None)`
 Makes new typed collection with optional new values.
 Returns new typed collection.

`TypedFrozenSet.__ne__(expr)`
 True when typed frozen set is not equal to *expr*. Otherwise false.
 Returns boolean.

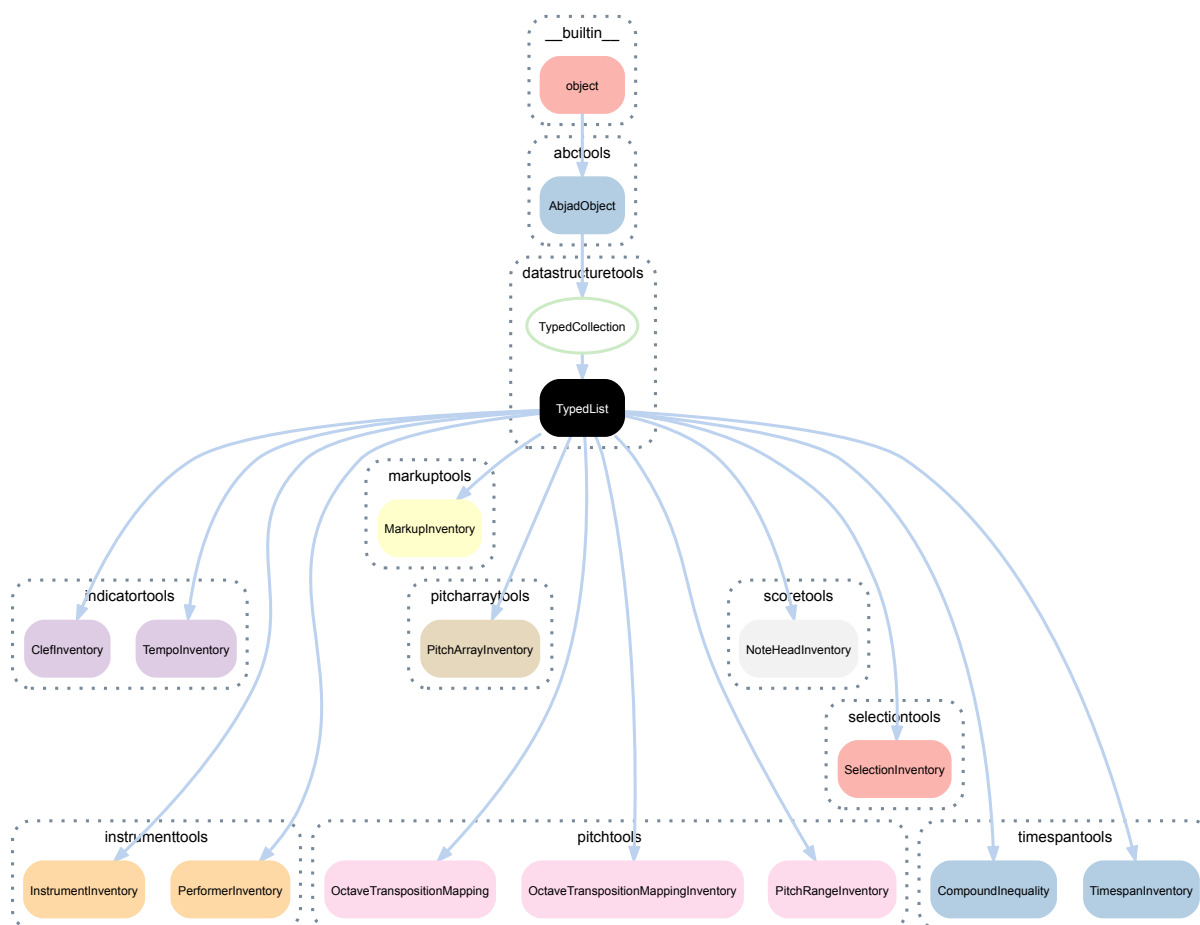
`TypedFrozenSet.__or__(expr)`
 Logical OR of typed frozen set and *expr*.
 Returns new typed frozen set.

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

`TypedFrozenSet.__sub__(expr)`
 Subtracts *expr* from typed frozen set.
 Returns new typed frozen set.

`TypedFrozenSet.__xor__(expr)`
 Logical XOR of typed frozen set and *expr*.
 Returns new typed frozen set.

2.2.15 datastructuretools.TypedList



class datastructuretools.**TypedList** (*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

A typed list.

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 format(object_collection)
datastructuretools.TypedList (
[
    23,
    'foo',
    False,
    (1, 2, 3),
    3.14159,
])
```

```
>>> print format(new(object_collection, keep_sorted=True))
datastructuretools.TypedList (
[
    23,
    'foo',
    False,
    (1, 2, 3),
    3.14159,
])
```

```
)

>>> pitch_collection = datastructuretools.TypedList (
...     item_class=NamedPitch)
>>> pitch_collection.append(0)
>>> pitch_collection.append("d' ")
>>> pitch_collection.append(('e', 4))
>>> pitch_collection.append(NamedPitch("f' "))
```

```
>>> print format(pitch_collection)
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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

`TypedList.keep_sorted`
Sorts collection on mutation if true.

Methods

`TypedList.append(token)`
Changes *token* to item and appends.

```
>>> 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)`

Changes *token* to item and returns 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)`

Changes *tokens* to items and extends.

```
>>> 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)`

Changes *token* to item and returns index.

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

Returns index.

`TypedList.insert(i, token)`

Changes *token* to item and inserts.

```
>>> 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.

`TypedList.pop(i=-1)`

Aliases `list.pop()`.

`TypedList.remove(token)`

Changes *token* to item and removes.

```
>>> 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)`
 True when typed collection container *token*. Otherwise false.
 Returns boolean.

`TypedList.__delitem__(i)`
 Aliases `list.__delitem__()`.

`(TypedCollection).__eq__(expr)`
 True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.
 Returns boolean.

`(TypedCollection).__format__(format_specification='')`
 Formats typed collection.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`TypedList.__getitem__(i)`
 Aliases `list.__getitem__()`.

`TypedList.__iadd__(expr)`
 Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList(
    [
        indicatortools.Dynamic(
            'ppp'
        ),
        indicatortools.Dynamic(
            'p'
        ),
        indicatortools.Dynamic(
            'mp'
        ),
        indicatortools.Dynamic(
            'mf'
        ),
        indicatortools.Dynamic(
            'fff'
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

`(TypedCollection).__iter__()`
 Iterates typed collection.
 Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

TypedList.**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

TypedList.**__reversed__**()

Aliases list.__reversed__().

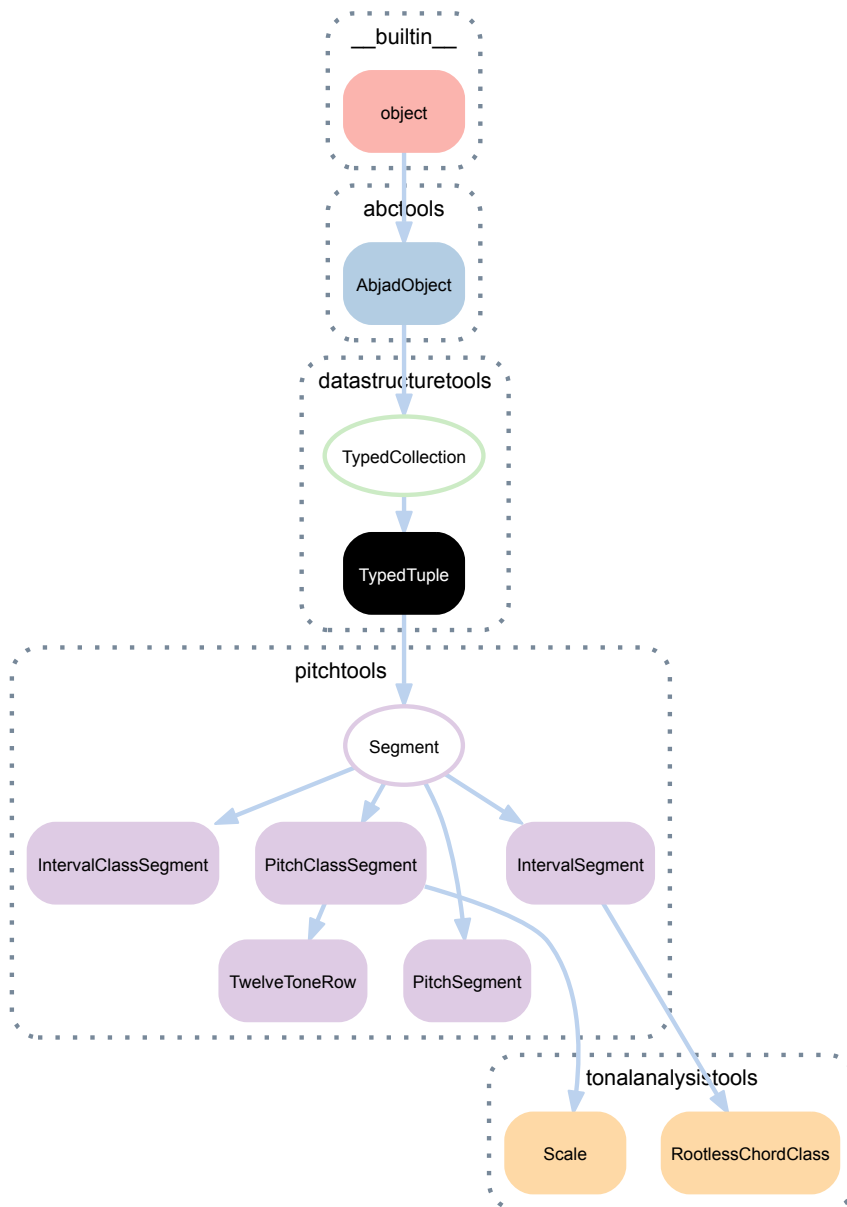
TypedList.**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```

2.2.16 datastructuretools.TypedTuple



class `datastructuretools.TypedTuple` (*tokens=None*, *item_class=None*, *custom_identifier=None*)
 A typed tuple.

Bases

- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

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

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

Methods

`TypedTuple.count(token)`
Changes *token* to item.
Returns count in collection.

`TypedTuple.index(token)`
Changes *token* to item.
Returns index in collection.

Special methods

`TypedTuple.__add__(expr)`
Adds typed tuple to *expr*.
Returns new typed tuple.

`TypedTuple.__contains__(token)`
Change *token* to item and return true if item exists in collection.
Returns none.

`(TypedCollection).__eq__(expr)`
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.
Returns boolean.

`(TypedCollection).__format__(format_specification='')`
Formats typed collection.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`TypedTuple.__getitem__(i)`
Gets *i* from type tuple.
Returns item.

`TypedTuple.__getslice__(start, stop)`
Gets slice from *start* to *stop* in typed tuple.
Returns new typed tuple.

`TypedTuple.__hash__()`
Hashes typed tuple.
Returns integer.

`(TypedCollection).__iter__()`
Iterates typed collection.
Returns generator.

`(TypedCollection).__len__()`
Length of typed collection.
Returns nonnegative integer.

(TypedCollection) .**__makenew__** (*tokens=None, item_class=None, custom_identifier=None*)

Makes new typed collection with optional new values.

Returns new typed collection.

TypedTuple .**__mul__** (*expr*)

Multiplies typed tuple by *expr*.

Returns new typed tuple.

(TypedCollection) .**__ne__** (*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

TypedTuple .**__rmul__** (*expr*)

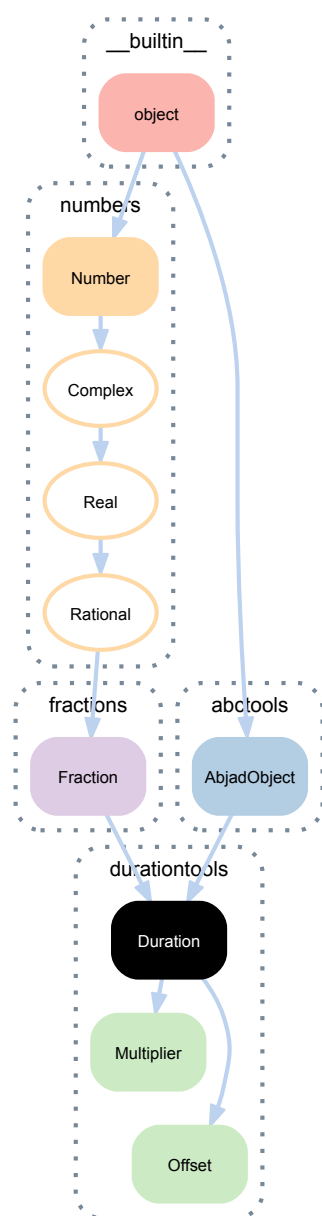
Multiplies *expr* by typed tuple.

Returns new typed tuple.

DURATIONTOOLS

3.1 Concrete classes

3.1.1 durationtools.Duration



`class durationtools.Duration`

A duration.

Initializes from integer numerator:

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

Initializes from integer numerator and denominator:

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

Initializes from integer-equivalent numeric numerator:

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

Initializes from integer-equivalent numeric numerator and denominator:

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

Initializes from integer-equivalent singleton:

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

Initializes from integer-equivalent pair:

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

Initializes from other duration:

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

Intialize from fraction:

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

Initializes from solidus string:

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

Initializes 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`

Number of dots required to notate duration.

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         sixteenths = duration.with_denominator(16)
...         dot_count = duration.dot_count
...         string = '{!s}\t{!s}'
...         string = string.format(sixteenths, dot_count)
...         print string
...     except AssignabilityError:
...         sixteenths = duration.with_denominator(16)
...         string = '{!s}\t{!s}'
...         string = string.format(sixteenths, '--')
...         print string
...
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
```

Raises assignability error when duration is not assignable.

Returns positive integer.

`Duration.equal_or_greater_assignable`

Assignable duration equal to or just greater than this duration.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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

Duration equal or just greater power of 2.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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

Assignable duration equal or just less than this duration.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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

Duration of the form $d \cdot 2$ equal to or just less than this duration.

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

```

...     print '{!s}\t{!s}'.format(sixteenths, 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

Number of flags required to notate duration.

```

>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     sixty_fourths = duration.with_denominator(64)
...     print '{!s}\t{}'.format(sixty_fourths, 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)
...     result = duration.has_power_of_two_denominator
...     print '{!s}\t{}'.format(duration, result)
...
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 duration.

```
>>> for denominator in range(1, 16 + 1):
...     duration = Duration(1, denominator)
...     result = duration.implied_prolation
...     print '{!s}\t{!s}'.format(duration, result)
...
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 duration is assignable. Otherwise false:

```
>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{}'.format(sixteenths, 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 duration.

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

Raises assignability error when duration is not assignable.

Returns string.

(Fraction) **.numerator**

Duration.pair

Duration numerator and denominator.

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

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

Returns integer pair.

Duration.prolation_string

Prolation string of duration.

```
>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     duration = generator.next()
...     string = '{!s}\t{!s}'.format(duration, duration.prolation_string)
...     print 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 new duration.

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)

Changes duration to clock string.

Changes numeric *seconds* to clock string:

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

Changes numeric *seconds* to escaped clock string:

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

```
>>> markup = markuptools.Markup('"%s"' % clock_string, Up)
>>> attach(markup, note)
```

Returns string.

Duration.with_denominator (*denominator*)

Change this duration to new 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.

Duration.yield_equivalent_durations (*minimum_written_duration=None*)

Yields all durations equivalent to this duration.

Returns output in Cantor diagonalized order.

Ensures written duration never less than *minimum_written_duration*.

Yields durations equivalent to 1/8:

```
>>> pairs = Duration(1, 8).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(1, 1), Duration(1, 8))
(Multiplier(2, 3), Duration(3, 16))
(Multiplier(4, 3), Duration(3, 32))
(Multiplier(4, 7), Duration(7, 32))
(Multiplier(8, 7), Duration(7, 64))
(Multiplier(8, 15), Duration(15, 64))
(Multiplier(16, 15), Duration(15, 128))
(Multiplier(16, 31), Duration(31, 128))
```

Yields durations equivalent to 1/12:

```
>>> pairs = Duration(1, 12).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(2, 3), Duration(1, 8))
(Multiplier(4, 3), Duration(1, 16))
(Multiplier(8, 9), Duration(3, 32))
(Multiplier(16, 9), Duration(3, 64))
(Multiplier(16, 21), Duration(7, 64))
(Multiplier(32, 21), Duration(7, 128))
(Multiplier(32, 45), Duration(15, 128))
```

Yields durations equivalent to 5/48:

```
>>> pairs = Duration(5, 48).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(5, 6), Duration(1, 8))
(Multiplier(5, 3), Duration(1, 16))
(Multiplier(5, 9), Duration(3, 16))
(Multiplier(10, 9), Duration(3, 32))
```

```
(Multiplier(20, 21), Duration(7, 64))
(Multiplier(40, 21), Duration(7, 128))
(Multiplier(8, 9), Duration(15, 128))
```

Defaults *minimum_written_duration* to 1/128.

Returns generator.

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**(*durations*)

Change *durations* to nonreduced fractions sharing least common denominator.

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

Returns new object of *durations* type.

Duration.**.from_lilypond_duration_string**(*lilypond_duration_string*)

Initializes duration from LilyPond duration string.

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

Returns duration.

Duration.**.is_token**(*expr*)

True when *expr* correctly initializes a duration. Otherwise false:

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

Returns boolean.

Duration.**.yield_durations**(*unique=False*)

Yields all positive durations.

Yields 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)
```

Yields 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)`
 Absolute value of duration.

Returns nonnegative duration.

`Duration.__add__(*args)`
 Adds duration to *args*.

Returns duration when *args* is a duration:

```
>>> duration_1 = Duration(1, 2)
>>> duration_2 = Duration(3, 2)
>>> duration_1 + duration_2
Duration(2, 1)
```

Returns nonreduced fraction when *args* is a nonreduced fraction:

```
>>> duration = Duration(1, 2)
>>> nonreduced_fraction = mathtools.NonreducedFraction(3, 6)
>>> duration + nonreduced_fraction
NonreducedFraction(6, 6)
```

Returns duration.

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

`(Fraction).__copy__()`

`(Fraction).__deepcopy__(memo)`

`Duration.__div__(*args)`
 Divides duration by *args*.

Returns multiplier.

`Duration.__divmod__(*args)`

Equals the pair (duration // *args*, duration % *args*).

Returns pair.

`Duration.__eq__(arg)`

True when duration equals *arg*. Otherwise false.

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`

`Duration.__format__(format_specification='')`

Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`Duration.__ge__(arg)`

True when duration is greater than or equal to *arg*. Otherwise false.

Returns boolean.

`Duration.__gt__(arg)`

True when duration is greater than *arg*. Otherwise 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.

`Duration.__le__(arg)`

True when duration is less than or equal to *arg*. Otherwise false.

Returns boolean.

`Duration.__lt__(arg)`

True when duration is less than *arg*. Otherwise false.

Returns boolean.

`Duration.__mod__(*args)`

Modulus operator applied to duration.

Returns duration.

`Duration.__mul__(*args)`

Duration multiplied by *args*.

Returns a new duration when *args* is a duration:

```
>>> duration_1 = Duration(1, 2)
>>> duration_2 = Duration(3, 2)
>>> duration_1 * duration_2
Duration(3, 4)
```

Returns nonreduced fraction when *args* is a nonreduced fraction:

```
>>> duration = Duration(1, 2)
>>> nonreduced_fraction = mathtools.NonreducedFraction(3, 6)
>>> duration * nonreduced_fraction
NonreducedFraction(3, 12)
```

Returns duration or nonreduced fraction.

`Duration.__ne__(arg)`

True when duration does not equal *arg*. Otherwise false.

Returns boolean.

`Duration.__neg__(*args)`

Negation of duration.

Returns duration.

`Duration.__new__(*args)`

`(Fraction).__nonzero__(a)`
`a != 0`

`Duration.__pos__(*args)`

Positive duration.

Returns duration.

`Duration.__pow__(*args)`

Duration raised to *args* power.

Returns duration.

`Duration.__radd__(*args)`

Adds *args* to duration.

Returns duration.

`Duration.__rdiv__(*args)`

Divides *args* by duration.

Returns duration.

`Duration.__rdivmod__(*args)`

Documentation required.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

`(Fraction).__rfloordiv__(b, a)`
`a // b`

`Duration.__rmod__(*args)`

Documentation required.

`Duration.__rmul__(*args)`

Multiplies *args* by duration.

Returns new duration.

`Duration.__rpow__(*args)`

Raises *args* to the power of duration.

Returns new duration.

`Duration.__rsub__(*args)`

Subtracts duration from *args*.

Returns new duration.

`Duration.__rtruediv__(*args)`

Documentation required.

Returns new duration.

`(Fraction).__str__()`

`str(self)`

`Duration.__sub__(*args)`

Subtracts *args* from duration.

Returns new duration.

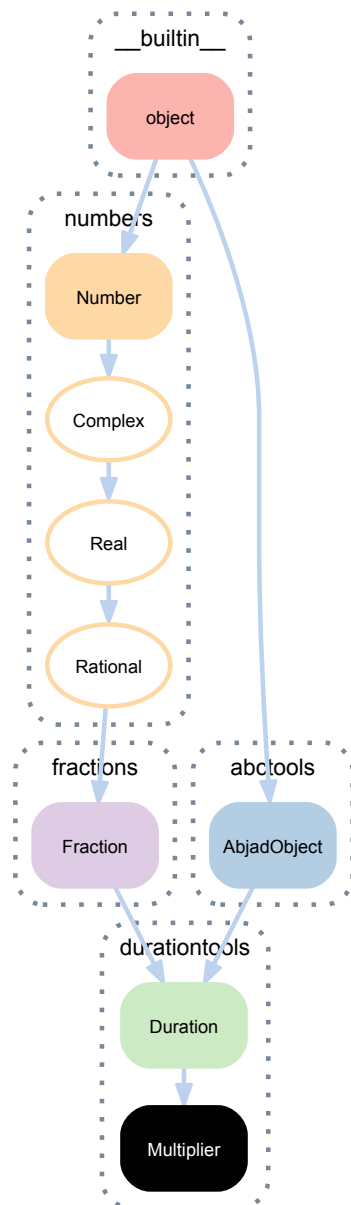
`Duration.__truediv__(*args)`

Documentation required.

`(Fraction).__trunc__(a)`

`trunc(a)`

3.1.2 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**

Number of dots required to notate duration.

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         sixteenths = duration.with_denominator(16)
...         dot_count = duration.dot_count
...         string = '{!s}\t{!s}'
...         string = string.format(sixteenths, dot_count)
...         print string
...     except AssignabilityError:
...         sixteenths = duration.with_denominator(16)
...         string = '{!s}\t{!s}'
...         string = string.format(sixteenths, '--')
...         print string
...
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
```

Raises assignability error when duration is not assignable.

Returns positive integer.

(`Duration`) **.equal_or_greater_assignable**

Assignable duration equal to or just greater than this duration.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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**
Duration equal or just greater power of 2.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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**
Assignable duration equal or just less than this duration.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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**

Duration of the form $d \cdot 2$ equal to or just less than this duration.

```

>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_power_of_two
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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**

Number of flags required to notate duration.

```

>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     sixty_fourths = duration.with_denominator(64)
...     print '{!s}\t{}'.format(sixty_fourths, 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)
...     result = duration.has_power_of_two_denominator
...     print '{!s}\t{}'.format(duration, result)
...

```

```

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 duration.

```

>>> for denominator in range(1, 16 + 1):
...     duration = Duration(1, denominator)
...     result = duration.implied_prolation
...     print '{!s}\t{!s}'.format(duration, result)
...
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 duration is assignable. Otherwise false:

```

>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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 duration.

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

Raises assignability error when duration is not assignable.

Returns string.

(Fraction).numerator

(Duration).pair

Duration numerator and denominator.

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

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

Returns integer pair.

(Duration).prolation_string

Prolation string of duration.

```
>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     duration = generator.next()
...     string = '{!s}\t{'
...     string = string.format(duration, duration.prolation_string)
...     print 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 new duration.

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*)
Changes duration to clock string.

Changes numeric *seconds* to clock string:

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

Changes numeric *seconds* to escaped clock string:

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

```
>>> markup = markuptools.Markup('"%s"' % clock_string, Up)
>>> attach(markup, note)
```

Returns string.

(Duration) **.with_denominator** (*denominator*)
Change this duration to new 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.

(Duration) **.yield_equivalent_durations** (*minimum_written_duration=None*)
Yields all durations equivalent to this duration.

Returns output in Cantor diagonalized order.

Ensures written duration never less than *minimum_written_duration*.

Yields durations equivalent to 1/8:

```
>>> pairs = Duration(1, 8).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(1, 1), Duration(1, 8))
(Multiplier(2, 3), Duration(3, 16))
(Multiplier(4, 3), Duration(3, 32))
(Multiplier(4, 7), Duration(7, 32))
(Multiplier(8, 7), Duration(7, 64))
(Multiplier(8, 15), Duration(15, 64))
(Multiplier(16, 15), Duration(15, 128))
(Multiplier(16, 31), Duration(31, 128))
```

Yields durations equivalent ot 1/12:

```
>>> pairs = Duration(1, 12).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(2, 3), Duration(1, 8))
(Multiplier(4, 3), Duration(1, 16))
(Multiplier(8, 9), Duration(3, 32))
(Multiplier(16, 9), Duration(3, 64))
(Multiplier(16, 21), Duration(7, 64))
(Multiplier(32, 21), Duration(7, 128))
(Multiplier(32, 45), Duration(15, 128))
```

Yields durations equivalent to $5/48$:

```
>>> pairs = Duration(5, 48).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(5, 6), Duration(1, 8))
(Multiplier(5, 3), Duration(1, 16))
(Multiplier(5, 9), Duration(3, 16))
(Multiplier(10, 9), Duration(3, 32))
(Multiplier(20, 21), Duration(7, 64))
(Multiplier(40, 21), Duration(7, 128))
(Multiplier(8, 9), Duration(15, 128))
```

Defaults *minimum_written_duration* to $1/128$.

Returns generator.

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**(*durations*)

Change *durations* to nonreduced fractions sharing least common denominator.

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

Returns new object of *durations* type.

(Duration).**from_lilypond_duration_string**(*lilypond_duration_string*)

Initializes duration from LilyPond duration string.

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

Returns duration.

(Duration).**is_token**(*expr*)

True when *expr* correctly initializes a duration. Otherwise false:

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

Returns boolean.

(Duration).**yield_durations** (*unique=False*)

Yields all positive durations.

Yields 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)
```

Yields 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)

Absolute value of duration.

Returns nonnegative duration.

(Duration).**__add__** (*args)

Adds duration to *args*.

Returns duration when *args* is a duration:

```
>>> duration_1 = Duration(1, 2)
>>> duration_2 = Duration(3, 2)
>>> duration_1 + duration_2
Duration(2, 1)
```

Returns nonreduced fraction when *args* is a nonreduced fraction:

```
>>> duration = Duration(1, 2)
>>> nonreduced_fraction = mathtools.NonreducedFraction(3, 6)
>>> duration + nonreduced_fraction
NonreducedFraction(6, 6)
```

Returns duration.

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

```
(Fraction) .__copy__()
```

```
(Fraction) .__deepcopy__(memo)
```

```
(Duration) .__div__(*args)
Divides duration by args.
```

Returns multiplier.

```
(Duration) .__divmod__(*args)
Equals the pair (duration // args, duration % args).
```

Returns pair.

```
(Duration) .__eq__(arg)
True when duration equals arg. Otherwise false.
```

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
```

```
(Duration) .__format__(format_specification='')
Formats duration.
```

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

```
(Duration) .__ge__(arg)
True when duration is greater than or equal to arg. Otherwise false.
```

Returns boolean.

```
(Duration) .__gt__(arg)
True when duration is greater than arg. Otherwise 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.

```
(Duration) .__le__(arg)
True when duration is less than or equal to arg. Otherwise false.
```

Returns boolean.

```
(Duration) .__lt__(arg)
True when duration is less than arg. Otherwise false.
```

Returns boolean.

(Duration) .**__mod__** (*args)
 Modulus operator applied to duration.
 Returns duration.

Multiplier.**__mul__** (*args)
 Multiplier times duration gives duration.
 Returns duration.

(Duration) .**__ne__** (arg)
 True when duration does not equal *arg*. Otherwise false.
 Returns boolean.

(Duration) .**__neg__** (*args)
 Negation of duration.
 Returns duration.

(Duration) .**__new__** (*args)

(Fraction) .**__nonzero__** (a)
 a != 0

(Duration) .**__pos__** (*args)
 Positive duration.
 Returns duration.

(Duration) .**__pow__** (*args)
 Duration raised to *args* power.
 Returns duration.

(Duration) .**__radd__** (*args)
 Adds *args* to duration.
 Returns duration.

(Duration) .**__rdiv__** (*args)
 Divides *args* by duration.
 Returns duration.

(Duration) .**__rdivmod__** (*args)
 Documentation required.

(AbjadObject) .**__repr__** ()
 Gets interpreter representation of Abjad object.
 Returns string.

(Fraction) .**__rfloordiv__** (b, a)
 a // b

(Duration) .**__rmod__** (*args)
 Documentation required.

(Duration) .**__rmul__** (*args)
 Multiplies *args* by duration.
 Returns new duration.

(Duration) .**__rpow__** (*args)
 Raises *args* to the power of duration.
 Returns new duration.

(Duration) .**__rsub__** (*args)
 Subtracts duration from *args*.

Returns new duration.

(Duration).**__rtruediv__**(*args)
Documentation required.

Returns new duration.

(Fraction).**__str__**()
str(self)

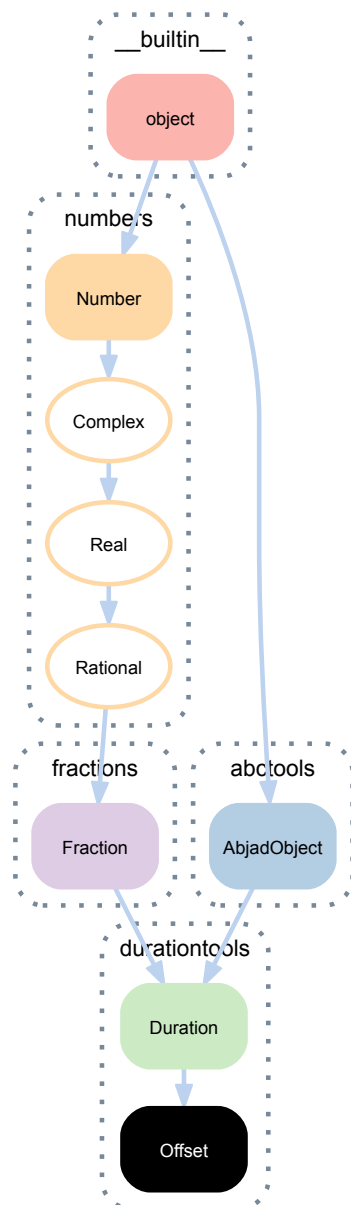
(Duration).**__sub__**(*args)
Subtracts *args* from duration.

Returns new duration.

(Duration).**__truediv__**(*args)
Documentation required.

(Fraction).**__trunc__**(a)
trunc(a)

3.1.3 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**

Number of dots required to notate duration.

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         sixteenths = duration.with_denominator(16)
...         dot_count = duration.dot_count
...         string = '{!s}\t{!s}'
...         string = string.format(sixteenths, dot_count)
...         print string
...     except AssignabilityError:
...         sixteenths = duration.with_denominator(16)
...         string = '{!s}\t{!s}'
...         string = string.format(sixteenths, '--')
...         print string
...
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
```

Raises assignability error when duration is not assignable.

Returns positive integer.

(Duration) **.equal_or_greater_assignable**

Assignable duration equal to or just greater than this duration.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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**

Duration equal or just greater power of 2.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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**

Assignable duration equal or just less than this duration.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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**

Duration of the form $d \cdot 2$ equal to or just less than this duration.

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_power_of_two
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{!s}'.format(sixteenths, 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**

Number of flags required to notate duration.

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     sixty_fourths = duration.with_denominator(64)
...     print '{!s}\t{}'.format(sixty_fourths, 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)
...     result = duration.has_power_of_two_denominator
...     print '{!s}\t{}'.format(duration, result)
...
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 duration.

```
>>> for denominator in range(1, 16 + 1):
...     duration = Duration(1, denominator)
...     result = duration.implied_prolation
...     print '{!s}\t{}'.format(duration, result)
...
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 duration is assignable. Otherwise false:

```
>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     sixteenths = duration.with_denominator(16)
...     print '{!s}\t{}'.format(sixteenths, 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 duration.

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

Raises assignability error when duration is not assignable.

Returns string.

(Fraction).**.numerator**

(Duration).**.pair**
Duration numerator and denominator.

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

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

Returns integer pair.

(Duration).**.prolation_string**
Prolation string of duration.

```

>>> generator = Duration.yield_durations(unique=True)
>>> for n in range(16):
...     duration = generator.next()
...     string = '{!s}\t{!s}'.format(duration, duration.prolation_string)
...     print 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 new duration.

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*)
Changes duration to clock string.

Changes numeric *seconds* to clock string:

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

Changes numeric *seconds* to escaped clock string:

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

```
>>> markup = markuptools.Markup('"%s"' % clock_string, Up)
>>> attach(markup, note)
```

Returns string.

(Duration) **.with_denominator** (*denominator*)
Change this duration to new 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.

(Duration) **.yield_equivalent_durations** (*minimum_written_duration=None*)
Yields all durations equivalent to this duration.

Returns output in Cantor diagonalized order.

Ensures written duration never less than *minimum_written_duration*.

Yields durations equivalent to 1/8:

```
>>> pairs = Duration(1, 8).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(1, 1), Duration(1, 8))
(Multiplier(2, 3), Duration(3, 16))
(Multiplier(4, 3), Duration(3, 32))
(Multiplier(4, 7), Duration(7, 32))
(Multiplier(8, 7), Duration(7, 64))
(Multiplier(8, 15), Duration(15, 64))
(Multiplier(16, 15), Duration(15, 128))
(Multiplier(16, 31), Duration(31, 128))
```

Yields durations equivalent to 1/12:

```
>>> pairs = Duration(1, 12).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(2, 3), Duration(1, 8))
(Multiplier(4, 3), Duration(1, 16))
(Multiplier(8, 9), Duration(3, 32))
(Multiplier(16, 9), Duration(3, 64))
(Multiplier(16, 21), Duration(7, 64))
(Multiplier(32, 21), Duration(7, 128))
(Multiplier(32, 45), Duration(15, 128))
```

Yields durations equivalent to $5/48$:

```
>>> pairs = Duration(5, 48).yield_equivalent_durations()
>>> for pair in pairs: pair
...
(Multiplier(5, 6), Duration(1, 8))
(Multiplier(5, 3), Duration(1, 16))
(Multiplier(5, 9), Duration(3, 16))
(Multiplier(10, 9), Duration(3, 32))
(Multiplier(20, 21), Duration(7, 64))
(Multiplier(40, 21), Duration(7, 128))
(Multiplier(8, 9), Duration(15, 128))
```

Defaults *minimum_written_duration* to $1/128$.

Returns generator.

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** (*durations*)

Change *durations* to nonreduced fractions sharing least common denominator.

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

Returns new object of *durations* type.

(Duration) **.from_lilypond_duration_string** (*lilypond_duration_string*)

Initializes duration from LilyPond duration string.

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

Returns duration.

(Duration) **.is_token** (*expr*)

True when *expr* correctly initializes a duration. Otherwise false:

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

Returns boolean.

(Duration).**yield_durations** (*unique=False*)

Yields all positive durations.

Yields 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)
```

Yields 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)

Absolute value of duration.

Returns nonnegative duration.

(Duration).**__add__** (*args)

Adds duration to *args*.

Returns duration when *args* is a duration:

```
>>> duration_1 = Duration(1, 2)
>>> duration_2 = Duration(3, 2)
>>> duration_1 + duration_2
Duration(2, 1)
```

Returns nonreduced fraction when *args* is a nonreduced fraction:

```
>>> duration = Duration(1, 2)
>>> nonreduced_fraction = mathtools.NonreducedFraction(3, 6)
>>> duration + nonreduced_fraction
NonreducedFraction(6, 6)
```

Returns duration.

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

```
(Fraction) .__copy__()
```

```
(Fraction) .__deepcopy__(memo)
```

```
(Duration) .__div__(*args)
Divides duration by args.
```

Returns multiplier.

```
(Duration) .__divmod__(*args)
Equals the pair (duration // args, duration % args).
```

Returns pair.

```
(Duration) .__eq__(arg)
True when duration equals arg. Otherwise false.
```

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
```

```
(Duration) .__format__(format_specification='')
Formats duration.
```

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

```
(Duration) .__ge__(arg)
True when duration is greater than or equal to arg. Otherwise false.
```

Returns boolean.

```
(Duration) .__gt__(arg)
True when duration is greater than arg. Otherwise 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.

```
(Duration) .__le__(arg)
True when duration is less than or equal to arg. Otherwise false.
```

Returns boolean.

```
(Duration) .__lt__(arg)
True when duration is less than arg. Otherwise false.
```

Returns boolean.

(Duration).**__mod__**(*args)
 Modulus operator applied to duration.
 Returns duration.

(Duration).**__mul__**(*args)
 Duration multiplied by *args*.
 Returns a new duration when *args* is a duration:

```
>>> duration_1 = Duration(1, 2)
>>> duration_2 = Duration(3, 2)
>>> duration_1 * duration_2
Duration(3, 4)
```

Returns nonreduced fraction when *args* is a nonreduced fraction:

```
>>> duration = Duration(1, 2)
>>> nonreduced_fraction = mathtools.NonreducedFraction(3, 6)
>>> duration * nonreduced_fraction
NonreducedFraction(3, 12)
```

Returns duration or nonreduced fraction.

(Duration).**__ne__**(arg)
 True when duration does not equal *arg*. Otherwise false.
 Returns boolean.

(Duration).**__neg__**(*args)
 Negation of duration.
 Returns duration.

(Duration).**__new__**(*args)

(Fraction).**__nonzero__**(a)
 a != 0

(Duration).**__pos__**(*args)
 Positive duration.
 Returns duration.

(Duration).**__pow__**(*args)
 Duration raised to *args* power.
 Returns duration.

(Duration).**__radd__**(*args)
 Adds *args* to duration.
 Returns duration.

(Duration).**__rdiv__**(*args)
 Divides *args* by duration.
 Returns duration.

(Duration).**__rdivmod__**(*args)
 Documentation required.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

(Fraction).**__rfloordiv__**(b, a)
 a // b

(Duration).**__rmod__**(*args)
 Documentation required.

(Duration) .**__rmul__**(*args)

Multiplies *args* by duration.

Returns new duration.

(Duration) .**__rpow__**(*args)

Raises *args* to the power of duration.

Returns new duration.

(Duration) .**__rsub__**(*args)

Subtracts duration from *args*.

Returns new duration.

(Duration) .**__rtruediv__**(*args)

Documentation required.

Returns new duration.

(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)

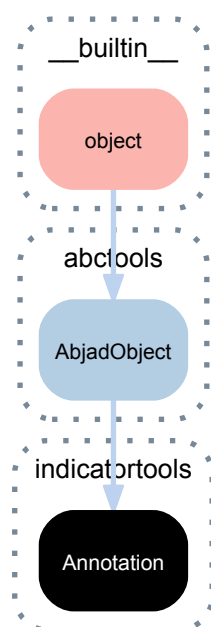
Documentation required.

(Fraction) .**__trunc__**(*a*)

trunc(a)

4.1 Concrete classes

4.1.1 `indicatortools.Annotation`



class `indicatortools.Annotation(*args)`
An annotation.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> pitch = NamedPitch('ds')
>>> annotation = indicatortools.Annotation('special pitch', pitch)
>>> attach(annotation, staff[0])
>>> show(staff)
```



Annotations contribute no formatting.

Bases

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

Read-only properties

`Annotation.name`

Name of annotation.

```
>>> annotation.name
'special pitch'
```

Returns string.

`Annotation.value`

Value of annotation.

```
>>> annotation.value
NamedPitch('ds')
```

Returns arbitrary object.

Special methods

`Annotation.__copy__(*args)`

Copies annotation.

Returns new annotation.

`Annotation.__eq__(arg)`

True when `arg` is an annotation with name and value equal to those of this annotation. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set `format_specification` to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal `expr`. Otherwise false.

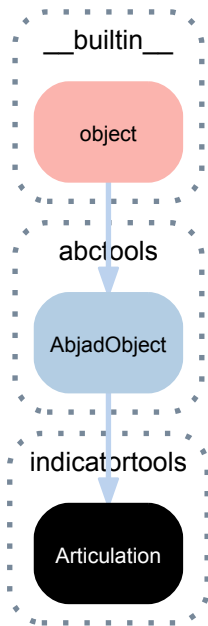
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.2 indicatortools.Articulation



class `indicatortools.Articulation(*args)`
An articulation.

Initializes from name:

```
>>> Articulation('staccato')
Articulation('staccato')
```

Initializes from abbreviation:

```
>>> Articulation('.')
Articulation('.')
```

Initializes from other articulation:

```
>>> articulation = Articulation('staccato')
>>> Articulation(articulation)
Articulation('staccato')
```

Initializes with direction:

```
>>> Articulation('staccato', Up)
Articulation('staccato', Up)
```

Use *attach()* to attach articulations to notes, rests or chords:

```
>>> note = Note("c'4")
>>> articulation = Articulation('staccato')
>>> attach(articulation, note)
>>> show(note)
```



Bases

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

Read-only properties

`Articulation.direction`

Direction of articulation.

Returns ordinal constant or none.

`Articulation.name`

Name of articulation.

```
>>> articulation.name
'staccato'
```

Returns string.

Special methods

`Articulation.__copy__(*args)`

Copies articulation.

Returns new articulation.

`Articulation.__eq__(expr)`

True when *expr* is an articulation with name and direction equal to that of this articulation. Otherwise false.

Returns boolean.

`Articulation.__format__(format_specification='')`

Formats articulation.

Set *format_specification* to '', 'lilypond' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`Articulation.__illustrate__()`

Illustrates articulation.

Returns LilyPond file.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

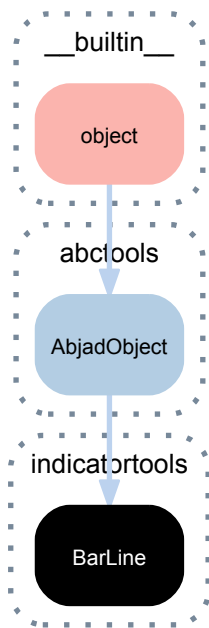
Returns string.

`Articulation.__str__()`

String representation of articulation.

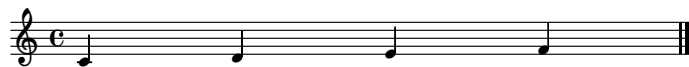
Returns string.

4.1.3 indicatortools.BarLine



class `indicatortools.BarLine` (*abbreviation='|'*)
A bar line.

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> bar_line = indicatortools.BarLine('|.')
>>> attach(bar_line, staff[-1])
>>> show(staff)
```



```
>>> bar_line
BarLine('|.')

```

Bases

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

Read-only properties

`BarLine.abbreviation`
Abbreviation of bar line.

```
>>> bar_line.abbreviation
'|.'
```

Returns string.

Special methods

`BarLine.__copy__(*args)`
Copies bar line.
Returns new bar line.

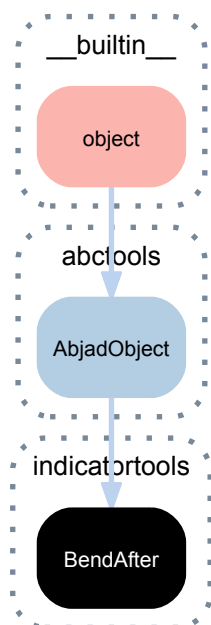
`BarLine.__eq__(arg)`
 True when *arg* is a bar line with an abbreviation equal to that of this bar line. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

4.1.4 indicatortools.BendAfter



class `indicatortools.BendAfter(*args)`
 A fall or doit.

```
>>> note = Note("c' 4")
>>> bend = indicatortools.BendAfter(-4)
>>> attach(bend, note)
>>> show(note)
```



Bases

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

Read-only properties

`BendAfter.bend_amount`
Amount of bend after.

```
>>> bend.bend_amount
-4.0
```

Returns float.

Special methods

`BendAfter.__copy__(*args)`
Copies bend after.

Returns new bend after.

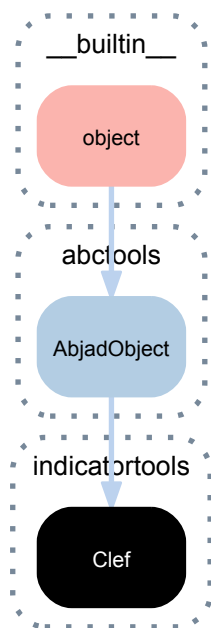
`BendAfter.__eq__(expr)`
True when *expr* is a bend after with bend amount equal to bend after. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

`BendAfter.__str__()`
String representation of bend after.
Returns string.

4.1.5 `indicatortools.Clef`

class `indicatortools.Clef` (*name*)

A clef.

```
>>> clef = Clef('treble')
>>> clef
Clef('treble')
```

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



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



Bases

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

Read-only properties

`Clef.middle_c_position`

Middle C position of clef.

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

Returns integer number of stafflines.

`Clef.name`

Name of clef.

Returns string.

Special methods

`Clef.__copy__(*args)`

Copies clef.

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

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

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

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

Returns new clef.

`Clef.__eq__(arg)`

True when clef name of *arg* equal clef name of clef. Otherwise false.

```
>>> clef_1 = Clef('treble')
>>> clef_2 = Clef('alto')
```

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

Returns boolean.

`Clef.__format__(format_specification='')`

Formats clef.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`. Interprets `'` equal to `'storage'`.

```
>>> clef = Clef('treble')
>>> print format(clef)
indicatortools.Clef(
    'treble'
)
```

Returns string.

`Clef.__ne__(arg)`

True when clef of *arg* does not equal clef name of clef. Otherwise false.

```
>>> clef_1 = Clef('treble')
>>> clef_2 = Clef('alto')
```

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

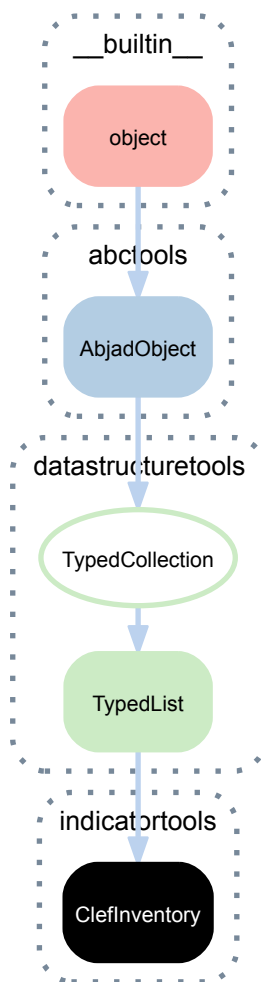
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

4.1.6 `indicatortools.ClefInventory`



class `indicatortools.ClefInventory` (*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

An ordered list of clefs.

```
>>> inventory = indicatortools.ClefInventory(['treble', 'bass'])
```

```
>>> inventory
ClefInventory([Clef('treble'), Clef('bass')])
```

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

```
>>> Clef('treble') in inventory
True
```

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

```
>>> show(inventory)
```



Clef 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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

`(TypedList).keep_sorted`
Sorts collection on mutation if true.

Methods

`(TypedList).append(token)`
Changes *token* to item and appends.

```
>>> 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)`
Changes *token* to item and returns 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*)
Changes *tokens* to items and extends.

```
>>> 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*)
Changes *token* to item and returns index.

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

Returns index.

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

```
>>> 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.

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

(TypedList) **.remove** (*token*)
Changes *token* to item and removes.

```
>>> 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)`
 True when typed collection container *token*. Otherwise false.
 Returns boolean.

`(TypedList).__delitem__(i)`
 Aliases `list.__delitem__()`.

`(TypedCollection).__eq__(expr)`
 True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.
 Returns boolean.

`(TypedCollection).__format__(format_specification='')`
 Formats typed collection.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`(TypedList).__getitem__(i)`
 Aliases `list.__getitem__()`.

`(TypedList).__iadd__(expr)`
 Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList(
[
    indicatortools.Dynamic(
        'ppp'
    ),
    indicatortools.Dynamic(
        'p'
    ),
    indicatortools.Dynamic(
        'mp'
    ),
    indicatortools.Dynamic(
        'mf'
    ),
    indicatortools.Dynamic(
        'fff'
    ),
],
item_class=indicatortools.Dynamic,
)
```

Returns collection.

`ClefInventory.__illustrate__()`
 Illustrates clef inventory.

```
>>> show(inventory)
```



Returns LilyPond file.

(TypedCollection).**__iter__**()
Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()
Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)
True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.

Returns string.

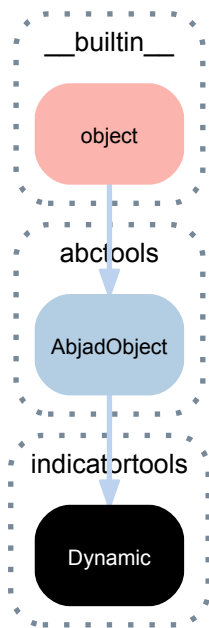
(TypedList).**__reversed__**()
Aliases list.**__reversed__**().

(TypedList).**__setitem__**(*i, expr*)
Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```


4.1.7 indicatortools.Dynamic



class `indicatortools.Dynamic` (*name*)
A dynamic.

Example 1. Initializes from dynamic name:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> dynamic = Dynamic('f')
>>> attach(dynamic, staff[0])
```

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



Example 2. Initializes from other dynamic:

```
>>> dynamic_1 = Dynamic('f')
>>> dynamic_2 = Dynamic(dynamic_1)
```

```
>>> dynamic_1
Dynamic('f')
```

```
>>> dynamic_2
Dynamic('f')
```

Bases

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

Read-only properties

`Dynamic.name`
Name of dynamic.

```
>>> dynamic.name
'f'
```

Returns string.

Static methods

`Dynamic.composite_dynamic_name_to_steady_state_dynamic_name(name)`
 Changes composite *name* to steady state dynamic name.

```
>>> Dynamic.composite_dynamic_name_to_steady_state_dynamic_name('sfp')
'p'
```

Returns string.

`Dynamic.dynamic_name_to_dynamic_ordinal(name)`
 Changes *name* to dynamic ordinal.

```
>>> Dynamic.dynamic_name_to_dynamic_ordinal('fff')
4
```

Returns integer.

`Dynamic.dynamic_ordinal_to_dynamic_name(dynamic_ordinal)`
 Changes *dynamic_ordinal* to dynamic name.

```
>>> Dynamic.dynamic_ordinal_to_dynamic_name(-5)
'pppp'
```

Returns string.

`Dynamic.is_dynamic_name(arg)`
 True when *arg* is dynamic name. Otherwise false.

```
>>> Dynamic.is_dynamic_name('f')
True
```

Returns boolean.

Special methods

`Dynamic.__copy__(*args)`
 Copies dynamic.

Returns new dynamic.

`Dynamic.__eq__(arg)`
 True when *arg* is a dynamic with a name equal to the name of this dynamic. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

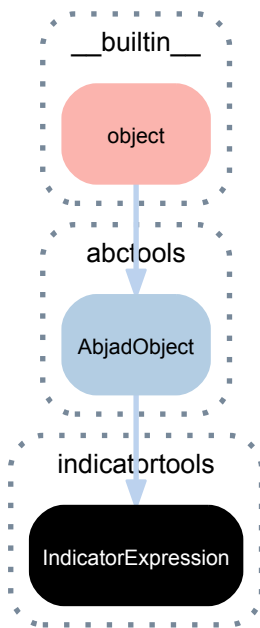
`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`
 Gets interpreter representation of Abjad object.

Returns string.

4.1.8 `indicatortools.IndicatorExpression`



class `indicatortools.IndicatorExpression` (*indicator=None, component=None, scope=None*)
 An indicator expression.

Bases

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

Read-only properties

`IndicatorExpression.component`
 Start component of indicator expression.
 Returns component.

`IndicatorExpression.indicator`
 Indicator of indicator expression.
 Returns indicator.

`IndicatorExpression.scope`
 Target context of indicator expression.
 Returns context.

Special methods

`IndicatorExpression.__copy__()`
 Copies indicator expression.

Note that indicator and scope are copied but that start component is not copied. This is to avoid start component reference problems.

Returns new indicator expression.

`IndicatorExpression.__eq__(arg)`

True when `arg` is an indicator expression with indicator and scope equal to those of this indicator expression. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set `format_specification` to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal `expr`. Otherwise false.

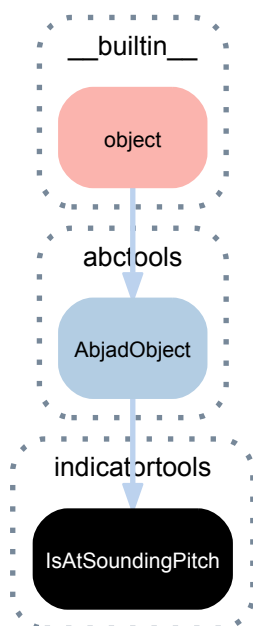
Returns boolean.

`IndicatorExpression.__repr__()`

Gets interpreter representation of indicator expression.

Returns string.

4.1.9 indicatortools.IsAtSoundingPitch



class `indicatortools.IsAtSoundingPitch`

Is at sounding pitch indicator.

```
>>> indicator = indicatortools.IsAtSoundingPitch()
```

Attach to score selection to denote music written at sounding pitch.

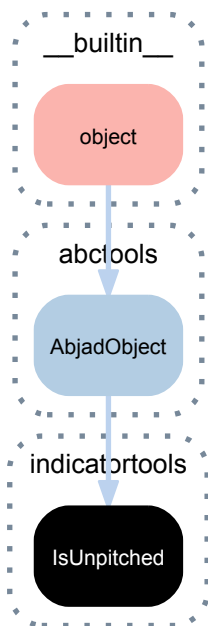
Bases

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

Special methods

- (AbjadObject).**__eq__**(*expr*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.
- (AbjadObject).**__format__**(*format_specification*='')
Formats object.
Set *format_specification* to ' or 'storage'. Interprets ' equal to 'storage'.
Returns string.
- (AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.
- (AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

4.1.10 indicatortools.IsUnpitched



class `indicatortools.IsUnpitched`
Is unpitched indicator.

```
>>> indicator = indicatortools.IsUnpitched()
```

Attach to score selection to denote unpitched music.

Bases

- `abctools.AbjadObject`
- `___builtin___object`

Special methods

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

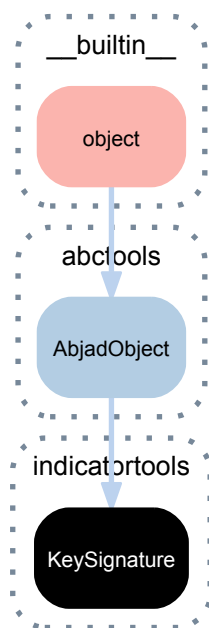
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

4.1.11 indicatortools.KeySignature



class `indicatortools.KeySignature` (*tonic*, *mode*)

A key signature.

```
>>> staff = Staff("e'8 fs'8 gs'8 a'8")
>>> key_signature = KeySignature('e', 'major')
>>> attach(key_signature, staff)
>>> show(staff)
```



Bases

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

Read-only properties

`KeySignature.mode`

Mode of signature.

```
>>> key_signature.mode
Mode('major')
```

Returns mode.

`KeySignature.name`

Name of key signature.

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

Returns string.

`KeySignature.tonic`

Tonic of key signature.

```
>>> key_signature.tonic
NamedPitchClass('e')
```

Returns named pitch-class.

Special methods

`KeySignature.__copy__(*args)`

Copies key signature.

Returns new key signature.

`KeySignature.__eq__(arg)`

True when *arg* is a key signature with tonic and mode equal to key signature. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

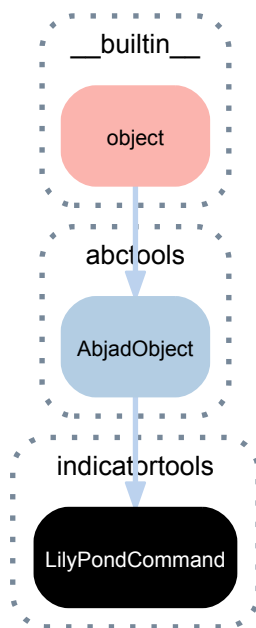
Returns string.

`KeySignature.__str__()`

String representation of key signature.

Returns string.

4.1.12 `indicatortools.LilyPondCommand`



class `indicatortools.LilyPondCommand` (*args)
A LilyPond command.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spanner.tools.Slur()
>>> attach(slur, staff.select_leaves())
```

```
>>> command = indicatortools.LilyPondCommand('slurDotted')
>>> attach(command, staff[0])
```

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



Initialize LilyPond commands from name; or from name with format slot; or from another LilyPond command; or from another LilyPond command with format slot.

Bases

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

Read-only properties

`LilyPondCommand.format_slot`
Format slot of LilyPond command.

```
>>> command.format_slot
'opening'
```

Returns string.

`LilyPondCommand.name`
Name of LilyPond command.


```
>>> command.name
'slurDotted'
```

Returns string.

Special methods

`LilyPondCommand.__copy__(*args)`
Copies LilyPond command.

Returns new LilyPond command.

`LilyPondCommand.__eq__(arg)`
True when *arg* is a LilyPond command with a name equal to that of this LilyPond command. Otherwise false.

Returns boolean.

`LilyPondCommand.__format__(format_specification='')`
Formats LilyPond command.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

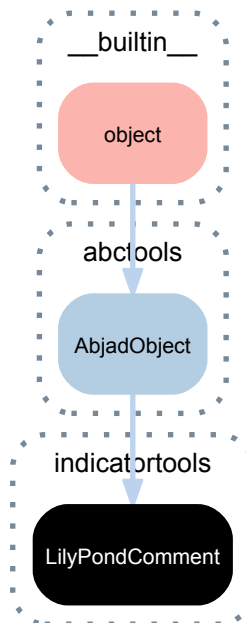
`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`
Gets interpreter representation of Abjad object.

Returns string.

4.1.13 `indicatortools.LilyPondComment`



class `indicatortools.LilyPondComment(*args)`
A LilyPond comment.

```
>>> note = Note("c'4")
>>> comment = indicatortools.LilyPondComment('this is a comment')
>>> attach(comment, note)
>>> show(note)
```



Initializes LilyPond comment from contents string; or contents string and format slot; or from other LilyPond comment; or from other LilyPond comment and format slot.

Bases

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

Read-only properties

`LilyPondComment.contents_string`
Contents string of LilyPond comment.

```
>>> comment.contents_string
'this is a comment'
```

Returns string.

`LilyPondComment.format_slot`
Format slot of LilyPond comment.

```
>>> comment.format_slot
'before'
```

Returns string.

Special methods

`LilyPondComment.__copy__(*args)`
Copies LilyPond comment.

Returns new LilyPond comment.

`LilyPondComment.__eq__(arg)`
True when *arg* is a LilyPond comment with contents string equal to LilyPond comment. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

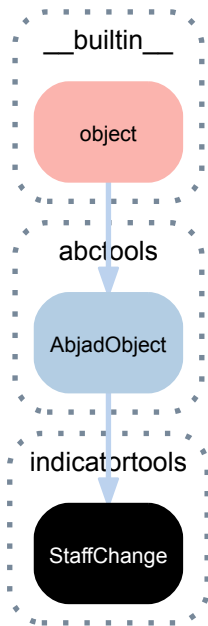
`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`
Gets interpreter representation of Abjad object.

Returns string.

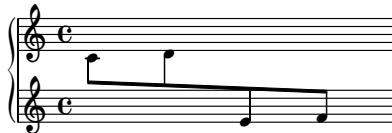
4.1.14 `indicatortools.StaffChange`



class `indicatortools.StaffChange` (*staff=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])
>>> staff_change = indicatortools.StaffChange(lh_staff)
>>> attach(staff_change, rh_staff[2])
>>> show(piano_staff)
  
```



Bases

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

Read-only properties

`StaffChange.staff`
 Staff of staff change.

```

>>> staff_change.staff
Staff-"LHStaff"{1}
  
```

Returns staff.

Special methods

`StaffChange.__copy__(*args)`

Copies staff change.

Returns new staff change.

`StaffChange.__eq__(arg)`

True when *arg* is a staff change with a staff value equal to that of this staff change. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

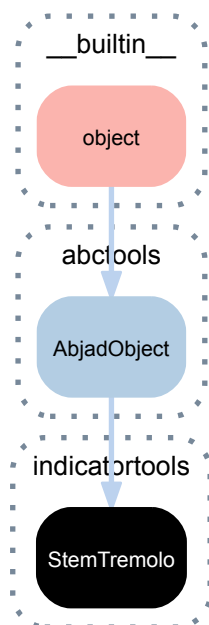
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.15 indicatortools.StemTremolo



class `indicatortools.StemTremolo(*args)`

A stem tremolo.

```

>>> note = Note("c'4")
>>> stem_tremolo = indicatortools.StemTremolo(16)
>>> attach(stem_tremolo, note)
>>> show(note)
  
```



Bases

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

Read-only properties

`StemTremolo.tremolo_flags`

Flags of stem tremolo.

```
>>> stem_tremolo.tremolo_flags
16
```

Returns nonnegative integer power of 2.

Special methods

`StemTremolo.__copy__(*args)`

Copies stem tremolo.

```
>>> import copy
>>> stem_tremolo_1 = indicatortools.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 = indicatortools.StemTremolo(16)
>>> stem_tremolo_2 = indicatortools.StemTremolo(16)
>>> stem_tremolo_3 = indicatortools.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.

`StemTremolo.__format__(format_specification='')`

Formats stem tremolo.

```
>>> print format(stem_tremolo)
:16
```

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

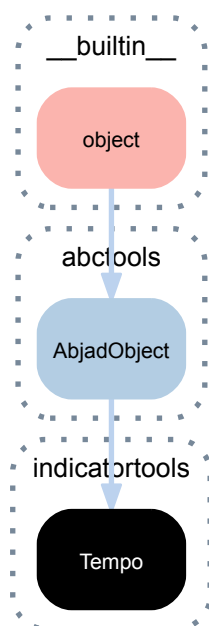
Returns string.

StemTremolo.**__str__**()

String representation of stem tremolo.

Returns string.

4.1.16 indicatortools.Tempo



class `indicatortools.Tempo(*args, **kwargs)`

A tempo indication.

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> tempo = Tempo(Duration(1, 8), 52)
>>> attach(tempo, staff[0])
>>> show(score)
```



Tempo indications are scoped to the **score context** by default.

Bases

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

Read-only properties

Tempo.**duration**

Duration of tempo.

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

Returns duration.

Tempo.**is_imprecise**

True if tempo is entirely textual or if tempo's `units_per_minute` is a range.

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

Otherwise false:

```
>>> Tempo(Duration(1, 4), 60).is_imprecise
False
```

Returns boolean.

Tempo.**quarters_per_minute**

Quarters per minute of tempo.

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

Returns tuple when tempo *units_per_minute* is a range.

Returns none when tempo is imprecise.

Returns fraction otherwise.

Tempo.**textual_indication**

Optional textual indication of tempo.

```
>>> tempo.textual_indication is None
True
```

Returns string or none.

Tempo.**units_per_minute**

Units per minute of tempo.

```
>>> tempo.units_per_minute
52
```

Returns number.

Methods

Tempo.**duration_to_milliseconds** (*duration*)

Millisecond value of *duration* under a given tempo.

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

Returns duration.

`Tempo.is_tempo_token(expr)`

True when *expr* can initialize tempo.

```
>>> tempo = Tempo(Duration(1, 4), 72)
>>> tempo.is_tempo_token((Duration(1, 4), 84))
True
```

Otherwise false:

```
>>> tempo.is_tempo_token(84)
False
```

Returns boolean.

`Tempo.list_related_tempos(maximum_numerator=None, maximum_denominator=None)`

Lists tempos related to this tempo.

Returns list of tempo / ratio pairs.

Each new tempo equals not less than half of this tempo and not more than twice this tempo.

Rewrites tempo 58 MM by ratios of the form $n:d$ such that $1 \leq n \leq 8$ and $1 \leq d \leq 8$: ...

```
>>> tempo = Tempo(Duration(1, 4), 58)
>>> pairs = tempo.list_related_tempos(
...     maximum_numerator=8,
...     maximum_denominator=8,
... )
```

```
>>> for tempo, ratio in pairs:
...     string = '{!s}\t{!s}'.format(tempo, ratio)
...     print string
4=29    1:2
4=58    1:1
4=87    3:2
4=116   2:1
```

Rewrites tempo 58 MM by ratios of the form $n:d$ such that $1 \leq n \leq 30$ and $1 \leq d \leq 30$:

```
>>> tempo = Tempo(Duration(1, 4), 58)
>>> pairs = tempo.list_related_tempos(
...     maximum_numerator=30,
...     maximum_denominator=30,
... )
```

```
>>> for tempo, ratio in pairs:
...     string = '{!s}\t{!s}'.format(tempo, ratio)
...     print string
...
4=30    15:29
4=32    16:29
4=34    17:29
4=36    18:29
4=38    19:29
4=40    20:29
4=42    21:29
4=44    22:29
4=46    23:29
4=48    24:29
4=50    25:29
4=52    26:29
4=54    27:29
4=56    28:29
4=58    1:1
4=60    30:29
```

Returns list.

`Tempo.rewrite_duration(duration, new_tempo)`

Rewrite *duration* under *new_tempo*.

Given *duration* governed by this tempo return new duration governed by *new_tempo*.

Ensure that *duration* and new duration consume the same amount of time in seconds.

Consider the two tempo indications below.

```
>>> tempo = Tempo(Duration(1, 4), 60)
>>> new_tempo = Tempo(Duration(1, 4), 90)
```

tempo specifies quarter equal to 60 MM.

new_tempo indication specifies quarter equal to 90 MM.

new_tempo is $3/2$ times as fast as *tempo*:

```
>>> new_tempo / tempo
Multiplier(3, 2)
```

Note that a triplet eighth note under *tempo* equals a regular eighth note under *new_tempo*:

```
>>> tempo.rewrite_duration(Duration(1, 12), new_tempo)
Duration(1, 8)
```

And note that a regular eighth note under *tempo* equals a dotted sixteenth under *new_tempo*:

```
>>> tempo.rewrite_duration(Duration(1, 8), new_tempo)
Duration(3, 16)
```

Returns duration.

Special methods

`Tempo.__add__(expr)`

Adds tempo to *expr*.

Returns new tempo.

`Tempo.__copy__(*args)`

Copies tempo.

Returns new tempo.

`Tempo.__div__(expr)`

Divides tempo by *expr*.

Returns new tempo.

`Tempo.__eq__(expr)`

True when *expr* is a tempo with duration, textual indication and units-per-minute all equal to this tempo. Otherwise false.

Returns boolean.

`Tempo.__format__(format_specification='')`

Formats tempo.

Set *format_specification* to `''`, `'lilypond'` or `'storage'`. Interprets `'` equal to `'storage'`.

```
>>> tempo = Tempo('Allegro', (1, 4), 84)
>>> print format(tempo)
indicatortools.Tempo(
    'Allegro',
    durationtools.Duration(1, 4),
    84
)
```

Returns string.

`Tempo.__ge__(other)`

`x.__ge__(y) <==> x>=y`

Tempo.__gt__(other)
 x.__gt__(y) <==> x>y

Tempo.__le__(other)
 x.__le__(y) <==> x<=y

Tempo.__lt__(arg)
 True when *arg* is a tempo with quarters per minute greater than that of this tempo. Otherwise false.
 Returns boolean.

Tempo.__mul__(multiplier)
 Multiplies tempo by *multiplier*.

```
>>> tempo = Tempo(Duration(1, 4), 84)
>>> tempo * 2
Tempo(Duration(1, 4), 168)
```

Returns new tempo.

(AbjadObject).__ne__(expr)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject).__repr__()
 Gets interpreter representation of Abjad object.
 Returns string.

Tempo.__rmul__(multiplier)
 Multiplies *multiplier* by tempo.

```
>>> tempo = Tempo(Duration(1, 4), 84)
>>> 2 * tempo
Tempo(Duration(1, 4), 168)
```

Returns new tempo.

Tempo.__str__()
 String representation of tempo.

```
>>> str(tempo)
'4=84'
```

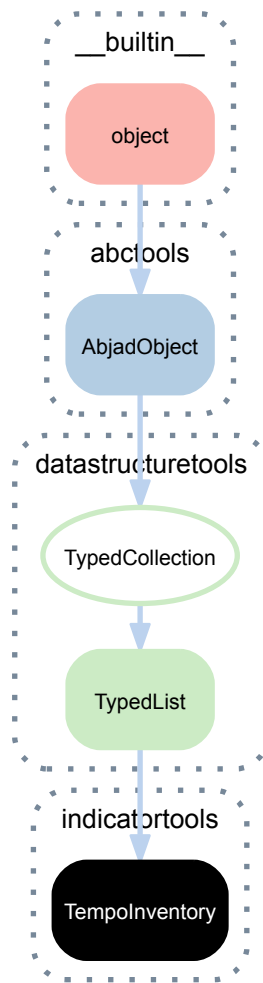
Returns string.

Tempo.__sub__(expr)
 Subtracts *expr* from tempo.

```
>>> tempo - 20
```

Returns new tempo.

4.1.17 indicatortools.TempoInventory



class `indicatortools.TempoInventory` (*tokens=None, item_class=None, keep_sorted=None, custom_idenfifier=None*)

An ordered list of tempo indications.

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

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

Tempo 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.

Read/write properties

(TypedCollection) .**custom_identifier**
Gets and sets custom identifier of typed collection.

Returns string or none.

(TypedList) .**keep_sorted**
Sorts collection on mutation if true.

Methods

(TypedList) .**append** (*token*)
Changes *token* to item and appends.

```
>>> 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*)
Changes *token* to item and returns 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*)
Changes *tokens* to items and extends.

```
>>> 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*)
Changes *token* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *token*)

Changes *token* to item and inserts.

```
>>> 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.

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

Aliases list.pop().

(TypedList) **.remove** (*token*)

Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) **.__format__** (*format_specification=''*)

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList).**__getitem__**(*i*)

Aliases list.**__getitem__**().

(TypedList).**__iadd__**(*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)  
datastructuretools.TypedList(  
    [  
        indicatortools.Dynamic(  
            'ppp'  
        ),  
        indicatortools.Dynamic(  
            'p'  
        ),  
        indicatortools.Dynamic(  
            'mp'  
        ),  
        indicatortools.Dynamic(  
            'mf'  
        ),  
        indicatortools.Dynamic(  
            'fff'  
        ),  
    ],  
    item_class=indicatortools.Dynamic,  
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, cus-*
tom_identifier=None)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList).**__reversed__**()

Aliases list.**__reversed__**().

(TypedList).**__setitem__**(*i, expr*)

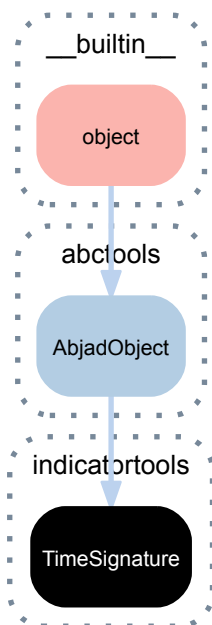
Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'  
>>> print format(pitch_collection)  
datastructuretools.TypedList(  
    [  
        'gqs',  
        'p',  
        'mp',  
        'mf',  
        'fff',  
    ],  
    item_class=indicatortools.Dynamic,  
)
```

```
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 format(pitch_collection)
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,
)
```

4.1.18 indicatortools.TimeSignature



class `indicatortools.TimeSignature(*args, **kwargs)`
 A time signature.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> time_signature = TimeSignature((4, 8))
>>> attach(time_signature, staff[0])
>>> show(staff)
```



Time signatures are scoped to the **staff** by default.

Set the scope of time signatures to the **score** like this:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> time_signature = TimeSignature((4, 8))
```

```
>>> attach(time_signature, staff[0], scope=Score)
>>> show(staff)
```



Bases

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

Read-only properties

`TimeSignature.denominator`

Time signature denominator.

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

Returns positive integer.

`TimeSignature.duration`

Time signature duration.

```
>>> TimeSignature((3, 8)).duration
Duration(3, 8)
```

Returns duration.

`TimeSignature.has_non_power_of_two_denominator`

True when time signature has non-power-of-two denominator.

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

Otherwise false:

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

Returns boolean.

`TimeSignature.implicit_prolation`

Time signature implied prolation.

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

```
>>> TimeSignature((3, 8)).implicit_prolation
Multiplier(1, 1)
```

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

```
>>> TimeSignature((7, 12)).implicit_prolation
Multiplier(2, 3)
```

Returns multiplier.

`TimeSignature.numerator`

Time signature numerator.

```
>>> time_signature.numerator
3
```

Returns positive integer.

`TimeSignature.pair`

Time signature numerator / denominator pair.

```
>>> TimeSignature((3, 8)).pair
(3, 8)
```

Returns pair.

`TimeSignature.partial`

Duration of time signature pick-up.

```
>>> time_signature.partial
```

Returns duration or none.

Methods

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

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

```
>>> time_signature = TimeSignature((3, 12))
```

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

Returns new time signature.

Special methods

`TimeSignature.__copy__(*args)`

Copies time signature.

Returns new time signature.

`TimeSignature.__eq__(arg)`

True when *arg* is a time signature with numerator and denominator equal to this time signature. Also true when *arg* is a tuple with first and second elements equal to numerator and denominator of this time signature. Otherwise false.

Returns boolean.

`TimeSignature.__format__(format_specification='')`

Formats time signature.

```
>>> print format(TimeSignature((3, 8)))
indicatortools.TimeSignature(
  (3, 8)
)
```

Returns string.

`TimeSignature.__ge__(arg)`

True when duration of time signature is greater than or equal to duration of *arg*. Otherwise false.

Returns boolean.

`TimeSignature.__gt__(arg)`

True when duration of time signature is greater than duration of *arg*. Otherwise false.

Returns boolean.

`TimeSignature.__le__(arg)`

True when duration of time signature is less than duration of *arg*. Otherwise false.

Returns boolean.

`TimeSignature.__lt__(arg)`

True when duration of time signature is less than duration of *arg*. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

`TimeSignature.__str__()`

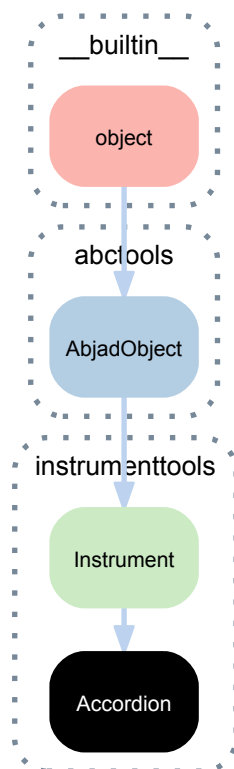
String representation of time signature.

Returns string.

INSTRUMENTTOOLS

5.1 Concrete classes

5.1.1 instrumenttools.Accordion



```
class instrumenttools.Accordion (instrument_name='accordion',
                                short_instrument_name='acc.',           in-
                                strument_name_markup=None,
                                short_instrument_name_markup=None,      allow-
                                able_clefs=('treble', 'bass'), pitch_range='[E1, C8]',
                                sounding_pitch_of_written_middle_c=None)
```

An accordion.

```
>>> piano_staff = scoretools.PianoStaff()
>>> piano_staff.append(Staff("c'4 d'4 e'4 f'4"))
>>> piano_staff.append(Staff("c'2 b2"))
>>> accordion = instrumenttools.Accordion()
>>> attach(accordion, piano_staff)
>>> attach(Clef('bass'), piano_staff[1])
>>> show(piano_staff)
```



The accordion targets the piano staff context by default.

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

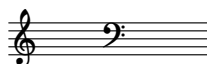
Read-only properties

`Accordion.allowable_clefs`

Gets accordion's allowable clefs.

```
>>> accordion.allowable_clefs
ClefInventory([Clef('treble'), Clef('bass')])
```

```
>>> show(accordion.allowable_clefs)
```



Returns clef inventory.

`Accordion.instrument_name`

Gets accordion's name.

```
>>> accordion.instrument_name
'accordion'
```

Returns string.

`Accordion.instrument_name_markup`

Gets accordion's instrument name markup.

```
>>> accordion.instrument_name_markup
Markup(('Accordion',))
```

```
>>> show(accordion.instrument_name_markup)
```

Accordion

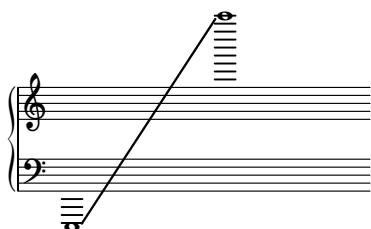
Returns markup.

`Accordion.pitch_range`

Gets accordion's range.

```
>>> accordion.pitch_range
PitchRange('E1, C8')
```

```
>>> show(accordion.pitch_range)
```



Returns pitch range.

`Accordion.short_instrument_name`

Gets accordion's short instrument name.

```
>>> accordion.short_instrument_name
'acc.'
```

Returns string.

`Accordion.short_instrument_name_markup`

Gets accordion's short instrument name markup.

```
>>> accordion.short_instrument_name_markup
Markup(('Acc.',))
```

```
>>> show(accordion.short_instrument_name_markup)
```

Acc.

Returns markup.

`Accordion.sounding_pitch_of_written_middle_c`

Gets sounding pitch of accordion's written middle C.

```
>>> accordion.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(accordion.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`Accordion.__format__(format_specification='')`

Formats accordion.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

```
>>> accordion = instrumenttools.Accordion()

>>> print format(accordion)
instrumenttools.Accordion(
  instrument_name='accordion',
  short_instrument_name='acc.',
  instrument_name_markup=markuptools.Markup(
    ('Accordion',)
  ),
  short_instrument_name_markup=markuptools.Markup(
    ('Acc.',)
  ),
  allowable_clefs=indicatortools.ClefInventory(
    [
      indicatortools.Clef(
        'treble'
```

```

        ),
        indicatortools.Clef(
            'bass'
        ),
    ]
),
pitch_range=pitchtools.PitchRange(
    '[E1, C8]'
),
sounding_pitch_of_written_middle_c=pitchtools.NamedPitch("c'"),
)

```

Returns string.

(Instrument).**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument).**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

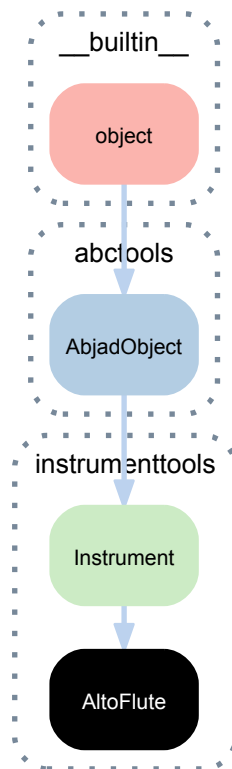
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

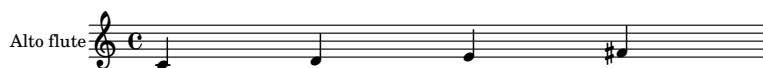
5.1.2 instrumenttools.AltoFlute



```
class instrumenttools.AltoFlute (instrument_name='alto flute', short_instrument_name='alt.
                                fl.', instrument_name_markup=None,
                                short_instrument_name_markup=None, allow-
                                able_clefs=None, pitch_range='[G3, G6]', sound-
                                ing_pitch_of_written_middle_c='G3')
```

An alto flute.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> alto_flute = instrumenttools.AltoFlute()
>>> attach(alto_flute, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

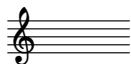
Read-only properties

`AltoFlute.allowable_clefs`

Gets alto flute's allowable clefs.

```
>>> alto_flute.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(alto_flute.allowable_clefs)
```



Returns clef inventory.

AltoFlute.instrument_name
Gets alto flute's name.

```
>>> alto_flute.instrument_name
'alto flute'
```

Returns string.

AltoFlute.instrument_name_markup
Gets alto flute's instrument name markup.

```
>>> alto_flute.instrument_name_markup
Markup(('Alto flute',))
```

```
>>> show(alto_flute.instrument_name_markup)
```

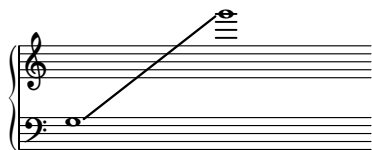
Alto flute

Returns markup.

AltoFlute.pitch_range
Gets alto flute's range.

```
>>> alto_flute.pitch_range
PitchRange(' [G3, G6]')
```

```
>>> show(alto_flute.pitch_range)
```



Returns pitch range.

AltoFlute.short_instrument_name
Gets alto flute's short instrument name.

```
>>> alto_flute.short_instrument_name
'alt. fl.'
```

Returns string.

AltoFlute.short_instrument_name_markup
Gets alto flute's short instrument name markup.

```
>>> alto_flute.short_instrument_name_markup
Markup(('Alt. fl.',))
```

```
>>> show(alto_flute.short_instrument_name_markup)
```

Alt. fl.

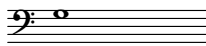
Returns markup.

AltoFlute.sounding_pitch_of_written_middle_c
Gets sounding pitch of alto flute's written middle C.

```
>>> alto_flute.sounding_pitch_of_written_middle_c
NamedPitch('g')
```



```
>>> show(alto_flute.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument).**__copy__**(*args)

Copies instrument.

Returns new instrument.

(Instrument).**__eq__**(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

AltoFlute.**__format__**(*format_specification*='')

Formats alto flute.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

```
>>> alto_flute = instrumenttools.AltoFlute()
>>> print format(alto_flute)
instrumenttools.AltoFlute(
  instrument_name='alto flute',
  short_instrument_name='alt. fl.',
  instrument_name_markup=markuptools.Markup(
    ('Alto flute',)
  ),
  short_instrument_name_markup=markuptools.Markup(
    ('Alt. fl.',)
  ),
  allowable_clefs=indicatortools.ClefInventory(
    [
      indicatortools.Clef(
        'treble'
      ),
    ]
  ),
  pitch_range=pitchtools.PitchRange(
    '[G3, G6]'
  ),
  sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('g'),
)
```

Returns string.

(Instrument).**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument).**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

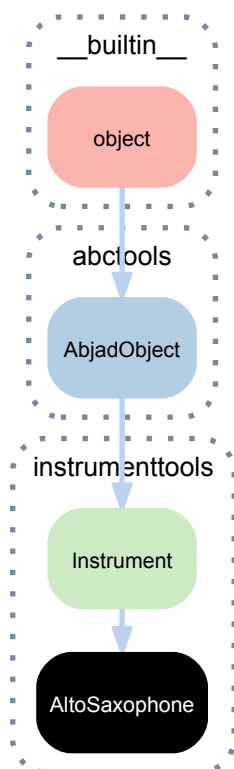
(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(Instrument).__repr__()`
 Gets interpreter representation of instrument.
 Returns string.

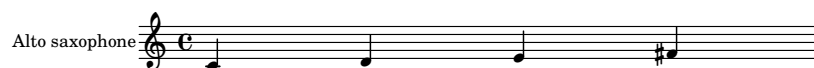
5.1.3 instrumenttools.AltoSaxophone



```
class instrumenttools.AltoSaxophone (instrument_name='alto saxophone',
                                     short_instrument_name='alt. sax.',
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,
                                     allowable_clefs=None, pitch_range='[Db3, A5]', sound-
                                     ing_pitch_of_written_middle_c='Eb3')
```

An alto saxophone.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> alto_saxophone = instrumenttools.AltoSaxophone()
>>> attach(alto_saxophone, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

AltoSaxophone.allowable_clefs

Gets alto saxophone's allowable clefs.

```
>>> alto_saxophone.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(alto_saxophone.allowable_clefs)
```



Returns clef inventory.

AltoSaxophone.instrument_name

Gets alto saxophone's name.

```
>>> alto_saxophone.instrument_name
'alto saxophone'
```

Returns string.

AltoSaxophone.instrument_name_markup

Gets alto saxophone's instrument name markup.

```
>>> alto_saxophone.instrument_name_markup
Markup(('Alto saxophone',))
```

```
>>> show(alto_saxophone.instrument_name_markup)
```

Alto saxophone

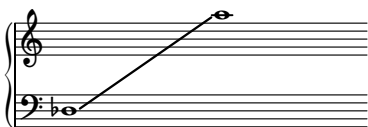
Returns markup.

AltoSaxophone.pitch_range

Gets alto saxophone's range.

```
>>> alto_saxophone.pitch_range
PitchRange('Db3, A5')
```

```
>>> show(alto_saxophone.pitch_range)
```



Returns pitch range.

AltoSaxophone.short_instrument_name

Gets alto saxophone's short instrument name.

```
>>> alto_saxophone.short_instrument_name
'alt. sax.'
```

Returns string.

AltoSaxophone.short_instrument_name_markup

Gets alto saxophone's short instrument name markup.

```
>>> alto_saxophone.short_instrument_name_markup
Markup(('Alt. sax.',))
```

```
>>> show(alto_saxophone.short_instrument_name_markup)
```

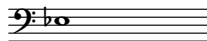
Alt. sax.

Returns markup.

`AltoSaxophone.sounding_pitch_of_written_middle_c`
 Gets sounding pitch of alto saxophone's written middle C.

```
>>> alto_saxophone.sounding_pitch_of_written_middle_c
NamedPitch('ef')
```

```
>>> show(alto_saxophone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`AltoSaxophone.__format__(format_specification='')`

Formats alto sax.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

```
>>> alto_sax = instrumenttools.AltoSaxophone()
>>> print format(alto_sax)
instrumenttools.AltoSaxophone(
  instrument_name='alto saxophone',
  short_instrument_name='alt. sax.',
  instrument_name_markup=markuptools.Markup(
    ('Alto saxophone',)
  ),
  short_instrument_name_markup=markuptools.Markup(
    ('Alt. sax.',)
  ),
  allowable_clefs=indicatortools.ClefInventory(
    [
      indicatortools.Clef(
        'treble'
      ),
    ]
  ),
  pitch_range=pitchtools.PitchRange(
    '[Db3, A5]'
  ),
  sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('ef'),
)
```

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

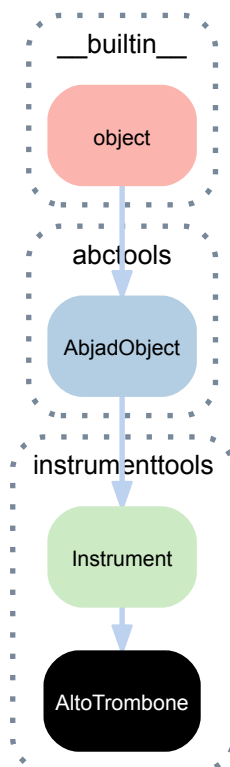
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.4 instrumenttools.AltoTrombone



```
class instrumenttools.AltoTrombone (instrument_name='alto',          trombone',
                                     short_instrument_name='alt.',   trb.',
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None, allow-
                                     able_clefs=('bass', 'tenor'), pitch_range='[A2, Bb5]',
                                     sounding_pitch_of_written_middle_c=None)
```

An alto trombone.

```
>>> staff = Staff("c4 d4 e4 fs4")
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> alto_trombone = instrumenttools.AltoTrombone()
>>> attach(alto_trombone, staff)
>>> show(staff)
```

Alto trombone

Bases

- `instrumenttools.Instrument`

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

Read-only properties

`AltoTrombone.allowable_clefs`

Gets alto trombone's allowable clefs.

```
>>> alto_trombone.allowable_clefs
ClefInventory([Clef('bass'), Clef('tenor')])
```

```
>>> show(alto_trombone.allowable_clefs)
```



Returns clef inventory.

`AltoTrombone.instrument_name`

Gets alto trombone's name.

```
>>> alto_trombone.instrument_name
'alto trombone'
```

Returns string.

`AltoTrombone.instrument_name_markup`

Gets alto trombone's instrument name markup.

```
>>> alto_trombone.instrument_name_markup
Markup(('Alto trombone',))
```

```
>>> show(alto_trombone.instrument_name_markup)
```

Alto trombone

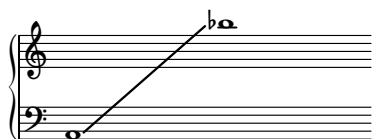
Returns markup.

`AltoTrombone.pitch_range`

Gets alto trombone's range.

```
>>> alto_trombone.pitch_range
PitchRange('A2, Bb5')
```

```
>>> show(alto_trombone.pitch_range)
```



Returns pitch range.

`AltoTrombone.short_instrument_name`

Gets alto trombone's short instrument name.

```
>>> alto_trombone.short_instrument_name
'alt. trb.'
```

Returns string.

`AltoTrombone.short_instrument_name_markup`

Gets alto trombone's short instrument name markup.

```
>>> alto_trombone.short_instrument_name_markup
Markup(('Alt. trb.',))
```

```
>>> show(alto_trombone.short_instrument_name_markup)
```

Alt. trb.

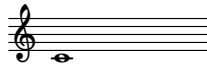
Returns markup.

`AltoTrombone.sounding_pitch_of_written_middle_c`

Gets sounding pitch of alto trombone's written middle C.

```
>>> alto_trombone.sounding_pitch_of_written_middle_c
NamedPitch("c")
```

```
>>> show(alto_trombone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`AltoTrombone.__format__(format_specification='')`

Formats alto trombone.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

```
>>> alto_trombone = instrumenttools.AltoTrombone()
>>> print format(alto_trombone)
instrumenttools.AltoTrombone(
  instrument_name='alto trombone',
  short_instrument_name='alt. trb.',
  instrument_name_markup=markuptools.Markup(
    ('Alto trombone',)
  ),
  short_instrument_name_markup=markuptools.Markup(
    ('Alt. trb.',)
  ),
  allowable_clefs=indicatortools.ClefInventory(
    [
      indicatortools.Clef(
        'bass'
      ),
      indicatortools.Clef(
        'tenor'
      ),
    ]
  ),
  pitch_range=pitchtools.PitchRange(
    '[A2, Bb5]'
  ),
  sounding_pitch_of_written_middle_c=pitchtools.NamedPitch("c"),
)
```

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

```
(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)
```

Makes new instrument.

Returns new instrument.

```
(AbjadObject).__ne__(expr)
```

Is true when Abjad object does not equal *expr*. Otherwise false.

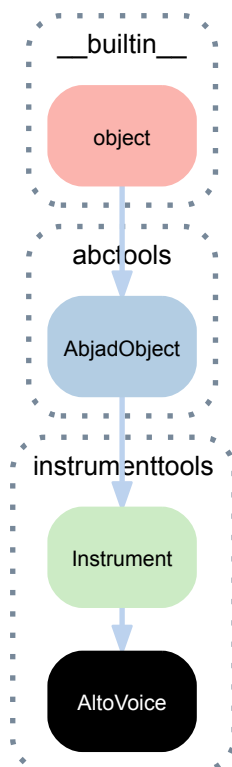
Returns boolean.

```
(Instrument).__repr__()
```

Gets interpreter representation of instrument.

Returns string.

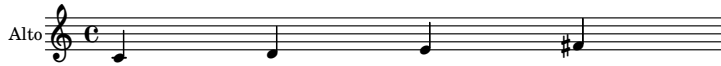
5.1.5 instrumenttools.AltoVoice



```
class instrumenttools.AltoVoice (instrument_name='alto', short_instrument_name='alto',
                                instrument_name_markup=None,
                                short_instrument_name_markup=None,
                                allowable_clefs=None, pitch_range='[F3, G5]',
                                sounding_pitch_of_written_middle_c=None)
```

A alto voice.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> alto = instrumenttools.AltoVoice()
>>> attach(alto, staff)
>>> show(staff)
```

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

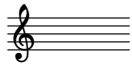
Read-only properties

`AltoVoice.allowable_clefs`

Gets alto's allowable clefs.

```
>>> alto.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(alto.allowable_clefs)
```



Returns clef inventory.

`AltoVoice.instrument_name`

Gets alto's name.

```
>>> alto.instrument_name
'alto'
```

Returns string.

`AltoVoice.instrument_name_markup`

Gets alto's instrument name markup.

```
>>> alto.instrument_name_markup
Markup(('Alto',))
```

```
>>> show(alto.instrument_name_markup)
```

Alto

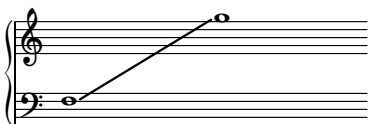
Returns markup.

`AltoVoice.pitch_range`

Gets alto's range.

```
>>> alto.pitch_range
PitchRange(' [F3, G5]')
```

```
>>> show(alto.pitch_range)
```



Returns pitch range.

`AltoVoice.short_instrument_name`

Gets alto's short instrument name.

```
>>> alto.short_instrument_name
'alto'
```

Returns string.

AltoVoice.short_instrument_name_markup
Gets alto's short instrument name markup.

```
>>> alto.short_instrument_name_markup
Markup(('Alto',))
```

```
>>> show(alto.short_instrument_name_markup)
```

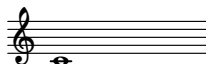
Alto

Returns markup.

AltoVoice.sounding_pitch_of_written_middle_c
Gets sounding pitch of alto's written middle C.

```
>>> alto.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(alto.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument).__copy__(*args)

Copies instrument.

Returns new instrument.

(Instrument).__eq__(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument).__format__(format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument).__hash__()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject).__ne__(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

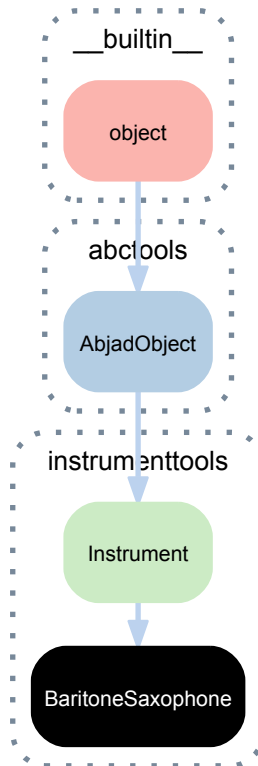
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.6 instrumenttools.BaritoneSaxophone



```
class instrumenttools.BaritoneSaxophone (instrument_name='baritone saxophone',
                                         short_instrument_name='bar. sax.',
                                         instrument_name_markup=None,
                                         short_instrument_name_markup=None,
                                         allowable_clefs=None, pitch_range='[C2, Ab4]',
                                         sounding_pitch_of_written_middle_c='Eb2')
```

A baritone saxophone.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> baritone_saxophone = instrumenttools.BaritoneSaxophone()
>>> attach(baritone_saxophone, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

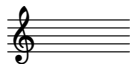
Read-only properties

BaritoneSaxophone.allowable_clefs

Gets baritone saxophone's allowable clefs.

```
>>> baritone_saxophone.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(baritone_saxophone.allowable_clefs)
```



Returns clef inventory.

BaritoneSaxophone.instrument_name

Gets baritone saxophone's name.

```
>>> baritone_saxophone.instrument_name
'baritone saxophone'
```

Returns string.

BaritoneSaxophone.instrument_name_markup

Gets baritone saxophone's instrument name markup.

```
>>> baritone_saxophone.instrument_name_markup
Markup(('Baritone saxophone',))
```

```
>>> show(baritone_saxophone.instrument_name_markup)
```

Baritone saxophone

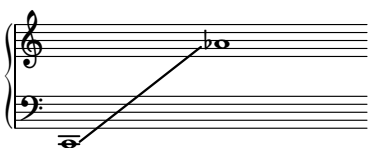
Returns markup.

BaritoneSaxophone.pitch_range

Gets baritone saxophone's range.

```
>>> baritone_saxophone.pitch_range
PitchRange(' [C2, Ab4]')
```

```
>>> show(baritone_saxophone.pitch_range)
```



Returns pitch range.

BaritoneSaxophone.short_instrument_name

Gets baritone saxophone's short instrument name.

```
>>> baritone_saxophone.short_instrument_name
'bar. sax.'
```

Returns string.

BaritoneSaxophone.short_instrument_name_markup

Gets baritone saxophone's short instrument name markup.

```
>>> baritone_saxophone.short_instrument_name_markup
Markup(('Bar. sax.',))
```

```
>>> show(baritone_saxophone.short_instrument_name_markup)
```

Bar. sax.

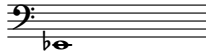
Returns markup.

`BaritoneSaxophone.sounding_pitch_of_written_middle_c`

Gets sounding pitch of baritone saxophone's written middle C.

```
>>> baritone_saxophone.sounding_pitch_of_written_middle_c
NamedPitch('ef,')
```

```
>>> show(baritone_saxophone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

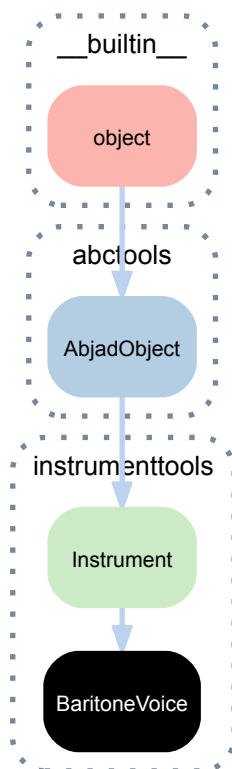
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.7 instrumenttools.BaritoneVoice



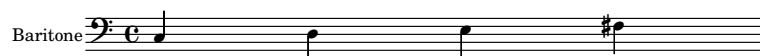
```

class instrumenttools.BaritoneVoice (instrument_name='baritone',
                                     short_instrument_name='bar.',          in-
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,      allow-
                                     able_clefs=('bass', ), pitch_range='[A2, A4]', sound-
                                     ing_pitch_of_written_middle_c=None)
  
```

A baritone voice.

```

>>> staff = Staff("c4 d4 e4 fs4")
>>> baritone = instrumenttools.BaritoneVoice()
>>> attach(baritone, staff)
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `___builtin___object`

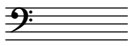
Read-only properties

`BaritoneVoice.allowable_clefs`
Gets baritone's allowable clefs.

```

>>> baritone.allowable_clefs
ClefInventory([Clef('bass')])
  
```

```
>>> show(baritone.allowable_clefs)
```



Returns clef inventory.

BaritoneVoice.instrument_name

Gets baritone's name.

```
>>> baritone.instrument_name
'baritone'
```

Returns string.

BaritoneVoice.instrument_name_markup

Gets baritone's instrument name markup.

```
>>> baritone.instrument_name_markup
Markup(('Baritone',))
```

```
>>> show(baritone.instrument_name_markup)
```

Baritone

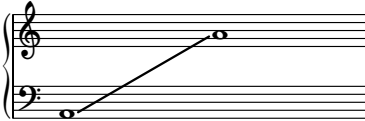
Returns markup.

BaritoneVoice.pitch_range

Gets baritone's range.

```
>>> baritone.pitch_range
PitchRange(' [A2, A4]')
```

```
>>> show(baritone.pitch_range)
```



Returns pitch range.

BaritoneVoice.short_instrument_name

Gets baritone's short instrument name.

```
>>> baritone.short_instrument_name
'bar.'
```

Returns string.

BaritoneVoice.short_instrument_name_markup

Gets baritone's short instrument name markup.

```
>>> baritone.short_instrument_name_markup
Markup(('Bar.',))
```

```
>>> show(baritone.short_instrument_name_markup)
```

Bar.

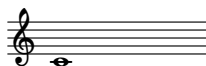
Returns markup.

BaritoneVoice.sounding_pitch_of_written_middle_c

Gets sounding pitch of baritone's written middle C.

```
>>> baritone.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(baron.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__ (*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__ (arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__ (format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__ ()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__ (instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__ (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

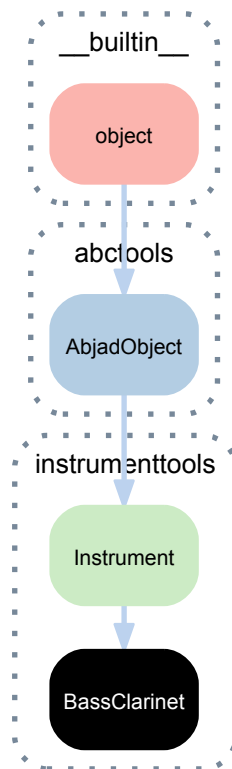
Returns boolean.

(Instrument) .__repr__ ()

Gets interpreter representation of instrument.

Returns string.

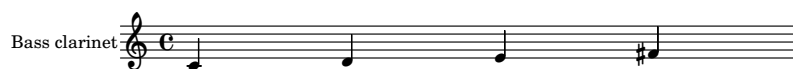
5.1.8 instrumenttools.BassClarinet



```
class instrumenttools.BassClarinet (instrument_name='bass', clarinet',
                                   short_instrument_name='bass', cl.',
                                   instrument_name_markup=None,
                                   short_instrument_name_markup=None, allow-
                                   able_clefs=('treble', 'bass'), pitch_range='[Bb1, G5]',
                                   sounding_pitch_of_written_middle_c='Bb2')
```

A bass clarinet.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> bass_clarinet = instrumenttools.BassClarinet()
>>> attach(bass_clarinet, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`BassClarinet.allowable_clefs`
Gets bass clarinet's allowable clefs.

```
>>> bass_clarinet.allowable_clefs
ClefInventory([Clef('treble'), Clef('bass')])
```

```
>>> show(bass_clarinet.allowable_clefs)
```



Returns clef inventory.

BassClarinet.instrument_name

Gets bass clarinet's name.

```
>>> bass_clarinet.instrument_name
'bass clarinet'
```

Returns string.

BassClarinet.instrument_name_markup

Gets bass clarinet's instrument name markup.

```
>>> bass_clarinet.instrument_name_markup
Markup(('Bass clarinet',))
```

```
>>> show(bass_clarinet.instrument_name_markup)
```

Bass clarinet

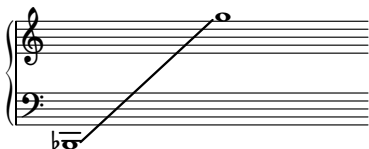
Returns markup.

BassClarinet.pitch_range

Gets bass clarinet's range.

```
>>> bass_clarinet.pitch_range
PitchRange(' [Bb1, G5]')
```

```
>>> show(bass_clarinet.pitch_range)
```



Returns pitch range.

BassClarinet.short_instrument_name

Gets bass clarinet's short instrument name.

```
>>> bass_clarinet.short_instrument_name
'bass cl.'
```

Returns string.

BassClarinet.short_instrument_name_markup

Gets bass clarinet's short instrument name markup.

```
>>> bass_clarinet.short_instrument_name_markup
Markup(('Bass cl.',))
```

```
>>> show(bass_clarinet.short_instrument_name_markup)
```

Bass cl.

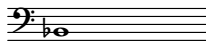
Returns markup.

BassClarinet.sounding_pitch_of_written_middle_c

Gets sounding pitch of bass_clarinet's written middle C.

```
>>> bass_clarinet.sounding_pitch_of_written_middle_c
NamedPitch('bf,')
```

```
>>> show(bass_clarinet.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

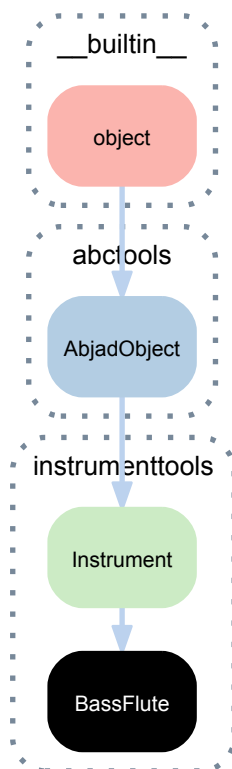
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

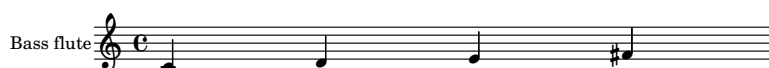
5.1.9 instrumenttools.BassFlute



```
class instrumenttools.BassFlute (instrument_name='bass flute', short_instrument_name='bass
                                fl.', instrument_name_markup=None,
                                short_instrument_name_markup=None, allow-
                                able_clefs=None, pitch_range='[C3, C6]', sound-
                                ing_pitch_of_written_middle_c='C3')
```

A bass flute.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> bass_flute = instrumenttools.BassFlute()
>>> attach(bass_flute, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

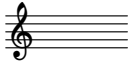
Read-only properties

`BassFlute.allowable_clefs`

Gets bass flute's allowable clefs.

```
>>> bass_flute.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(bass_flute.allowable_clefs)
```



Returns clef inventory.

BassFlute.instrument_name

Gets bass flute's name.

```
>>> bass_flute.instrument_name
'bass flute'
```

Returns string.

BassFlute.instrument_name_markup

Gets bass flute's instrument name markup.

```
>>> bass_flute.instrument_name_markup
Markup(('Bass flute',))
```

```
>>> show(bass_flute.instrument_name_markup)
```

Bass flute

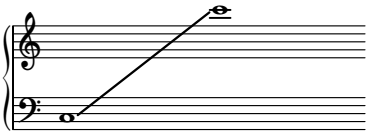
Returns markup.

BassFlute.pitch_range

Gets bass flute's range.

```
>>> bass_flute.pitch_range
PitchRange(' [C3, C6]')
```

```
>>> show(bass_flute.pitch_range)
```



Returns pitch range.

BassFlute.short_instrument_name

Gets bass flute's short instrument name.

```
>>> bass_flute.short_instrument_name
'bass fl.'
```

Returns string.

BassFlute.short_instrument_name_markup

Gets bass flute's short instrument name markup.

```
>>> bass_flute.short_instrument_name_markup
Markup(('Bass fl.',))
```

```
>>> show(bass_flute.short_instrument_name_markup)
```

Bass fl.

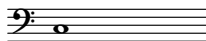
Returns markup.

BassFlute.sounding_pitch_of_written_middle_c

Gets sounding pitch of bass_flute's written middle C.

```
>>> bass_flute.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(bass_flute.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

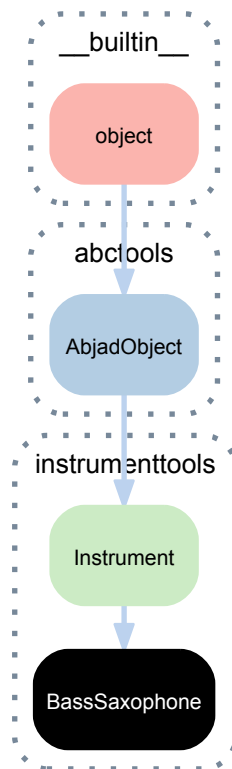
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.10 instrumenttools.BassSaxophone



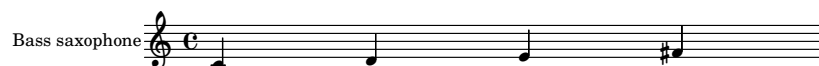
```

class instrumenttools.BassSaxophone (instrument_name='bass saxophone',
                                     short_instrument_name='bass sax.',
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,
                                     allowable_clefs=None, pitch_range='[Ab2, E4]',
                                     sounding_pitch_of_written_middle_c='Bb1')
  
```

A bass saxophone.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> bass_saxophone = instrumenttools.BassSaxophone()
>>> attach(bass_saxophone, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

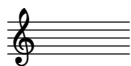
Read-only properties

`BassSaxophone.allowable_clefs`
Gets bass saxophone's allowable clefs.

```

>>> bass_saxophone.allowable_clefs
ClefInventory([Clef('treble')])
  
```

```
>>> show(bass_saxophone.allowable_clefs)
```



Returns clef inventory.

BassSaxophone.instrument_name

Gets bass saxophone's name.

```
>>> bass_saxophone.instrument_name
'bass saxophone'
```

Returns string.

BassSaxophone.instrument_name_markup

Gets bass saxophone's instrument name markup.

```
>>> bass_saxophone.instrument_name_markup
Markup(('Bass saxophone',))
```

```
>>> show(bass_saxophone.instrument_name_markup)
```

Bass saxophone

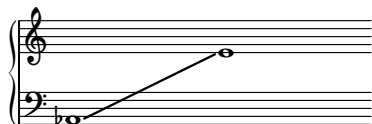
Returns markup.

BassSaxophone.pitch_range

Gets bass saxophone's range.

```
>>> bass_saxophone.pitch_range
PitchRange(' [Ab2, E4]')
```

```
>>> show(bass_saxophone.pitch_range)
```



Returns pitch range.

BassSaxophone.short_instrument_name

Gets bass saxophone's short instrument name.

```
>>> bass_saxophone.short_instrument_name
'bass sax.'
```

Returns string.

BassSaxophone.short_instrument_name_markup

Gets bass saxophone's short instrument name markup.

```
>>> bass_saxophone.short_instrument_name_markup
Markup(('Bass sax.',))
```

```
>>> show(bass_saxophone.short_instrument_name_markup)
```

Bass sax.

Returns markup.

BassSaxophone.sounding_pitch_of_written_middle_c

Gets sounding pitch of bass_saxophone's written middle C.

```
>>> bass_saxophone.sounding_pitch_of_written_middle_c
NamedPitch('bf,,')
```



```
>>> show(bass_saxophone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

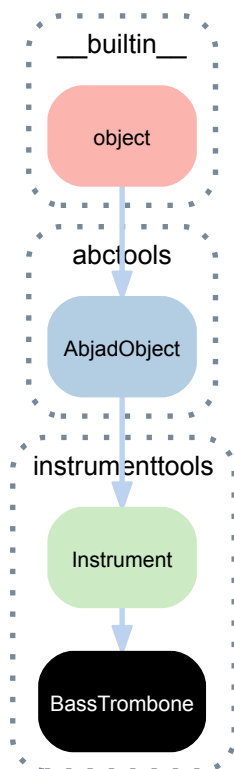
Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

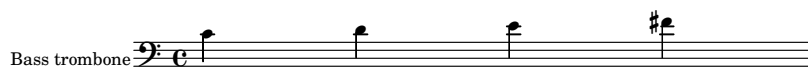
5.1.11 instrumenttools.BassTrombone



```
class instrumenttools.BassTrombone (instrument_name='bass          trombone',
                                   short_instrument_name='bass          trb.',
                                   instrument_name_markup=None,
                                   short_instrument_name_markup=None,
                                   allowable_clefs=('bass', ), pitch_range='[C2, F4]', sound-
                                   ing_pitch_of_written_middle_c=None)
```

A bass trombone.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> bass_trombone = instrumenttools.BassTrombone()
>>> attach(bass_trombone, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`BassTrombone.allowable_clefs`
Gets bass trombone's allowable clefs.

```
>>> bass_trombone.allowable_clefs
ClefInventory([Clef('bass')])
```

```
>>> show(bass_trombone.allowable_clefs)
```



Returns clef inventory.

BassTrombone.instrument_name

Gets bass trombone's name.

```
>>> bass_trombone.instrument_name
'bass trombone'
```

Returns string.

BassTrombone.instrument_name_markup

Gets bass trombone's instrument name markup.

```
>>> bass_trombone.instrument_name_markup
Markup(('Bass trombone',))
```

```
>>> show(bass_trombone.instrument_name_markup)
```

Bass trombone

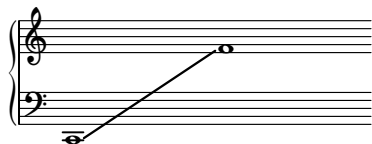
Returns markup.

BassTrombone.pitch_range

Gets bass trombone's range.

```
>>> bass_trombone.pitch_range
PitchRange(' [C2, F4]')
```

```
>>> show(bass_trombone.pitch_range)
```



Returns pitch range.

BassTrombone.short_instrument_name

Gets bass trombone's short instrument name.

```
>>> bass_trombone.short_instrument_name
'bass trb.'
```

Returns string.

BassTrombone.short_instrument_name_markup

Gets bass trombone's short instrument name markup.

```
>>> bass_trombone.short_instrument_name_markup
Markup(('Bass trb.',))
```

```
>>> show(bass_trombone.short_instrument_name_markup)
```

Bass trb.

Returns markup.

BassTrombone.sounding_pitch_of_written_middle_c

Gets sounding pitch of bass_trombone's written middle C.

```
>>> bass_trombone.sounding_pitch_of_written_middle_c
NamedPitch("c")
```

```
>>> show(bass_trombone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument).**__copy__**(*args)

Copies instrument.

Returns new instrument.

(Instrument).**__eq__**(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument).**__format__**(format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument).**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument).**__makenew__**(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject).**__ne__**(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

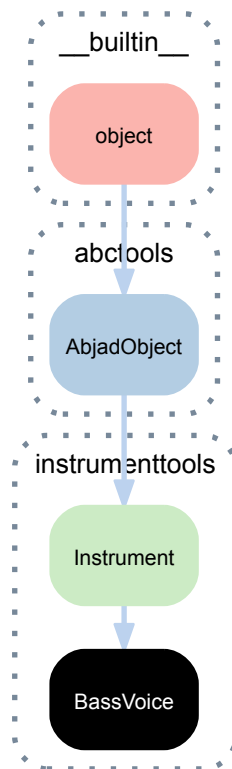
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.12 instrumenttools.BassVoice



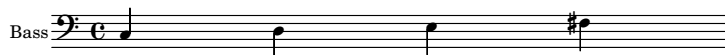
```

class instrumenttools.BassVoice (instrument_name='bass',    short_instrument_name='bass',
                                instrument_name_markup=None,
                                short_instrument_name_markup=None,    allow-
                                able_clefs=('bass', ),    pitch_range='[E2, F4]',    sound-
                                ing_pitch_of_written_middle_c=None)
  
```

A bass.

```

>>> staff = Staff("c4 d4 e4 fs4")
>>> bass = instrumenttools.BassVoice()
>>> attach(bass, staff)
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

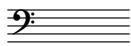
`BassVoice.allowable_clefs`

Gets bass's allowable clefs.

```

>>> bass.allowable_clefs
ClefInventory([Clef('bass')])
  
```

```
>>> show(bass.allowable_clefs)
```



Returns clef inventory.

BassVoice.instrument_name
Gets bass's name.

```
>>> bass.instrument_name
'bass'
```

Returns string.

BassVoice.instrument_name_markup
Gets bass's instrument name markup.

```
>>> bass.instrument_name_markup
Markup(('Bass',))
```

```
>>> show(bass.instrument_name_markup)
```

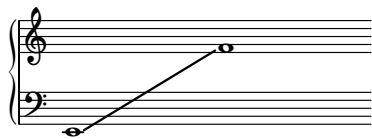
Bass

Returns markup.

BassVoice.pitch_range
Gets bass's range.

```
>>> bass.pitch_range
PitchRange(' [E2, F4]')
```

```
>>> show(bass.pitch_range)
```



Returns pitch range.

BassVoice.short_instrument_name
Gets bass's short instrument name.

```
>>> bass.short_instrument_name
'bass'
```

Returns string.

BassVoice.short_instrument_name_markup
Gets bass's short instrument name markup.

```
>>> bass.short_instrument_name_markup
Markup(('Bass',))
```

```
>>> show(bass.short_instrument_name_markup)
```

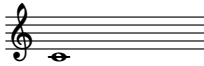
Bass

Returns markup.

BassVoice.sounding_pitch_of_written_middle_c
Gets sounding pitch of bass's written middle C.

```
>>> bass.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(bass.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

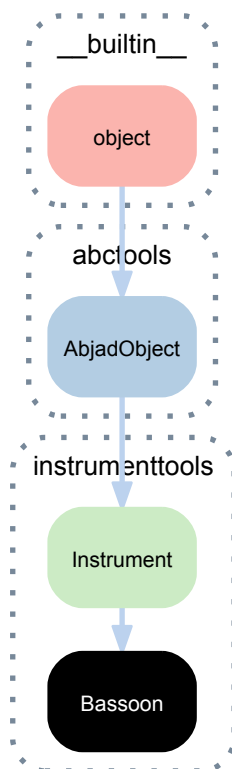
Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

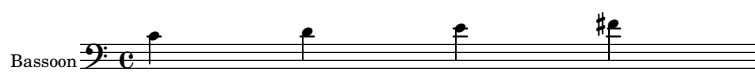
5.1.13 instrumenttools.Bassoon



```
class instrumenttools.Bassoon(instrument_name='bassoon', short_instrument_name='bsn.',
                              instrument_name_markup=None,
                              short_instrument_name_markup=None, allow-
                              able_clefs=('bass', 'tenor'), pitch_range='[Bb1, Eb5]',
                              sounding_pitch_of_written_middle_c=None)
```

A bassoon.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> bassoon = instrumenttools.Bassoon()
>>> attach(bassoon, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

Bassoon.allowable_clefs
Gets bassoon's allowable clefs.

```
>>> bassoon.allowable_clefs
ClefInventory([Clef('bass'), Clef('tenor')])
```



```
>>> show(bassoon.allowable_clefs)
```



Returns clef inventory.

Bassoon.instrument_name

Gets bassoon's name.

```
>>> bassoon.instrument_name
'bassoon'
```

Returns string.

Bassoon.instrument_name_markup

Gets bassoon's instrument name markup.

```
>>> bassoon.instrument_name_markup
Markup(('Bassoon',))
```

```
>>> show(bassoon.instrument_name_markup)
```

Bassoon

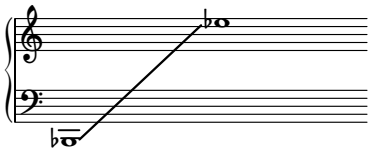
Returns markup.

Bassoon.pitch_range

Gets bassoon's range.

```
>>> bassoon.pitch_range
PitchRange(' [Bb1, Eb5]')
```

```
>>> show(bassoon.pitch_range)
```



Returns pitch range.

Bassoon.short_instrument_name

Gets bassoon's short instrument name.

```
>>> bassoon.short_instrument_name
'bsn.'
```

Returns string.

Bassoon.short_instrument_name_markup

Gets bassoon's short instrument name markup.

```
>>> bassoon.short_instrument_name_markup
Markup(('Bsn.',))
```

```
>>> show(bassoon.short_instrument_name_markup)
```

Bsn.

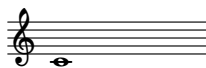
Returns markup.

Bassoon.sounding_pitch_of_written_middle_c

Gets sounding pitch of bassoon's written middle C.

```
>>> bassoon.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(bassoon.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

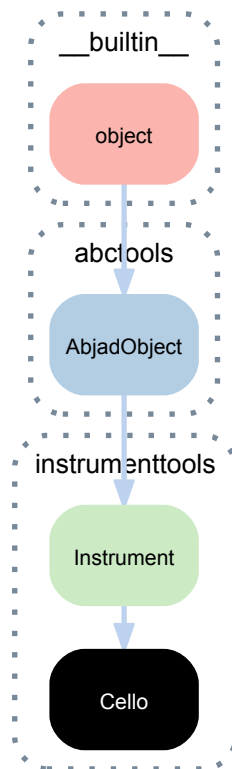
Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

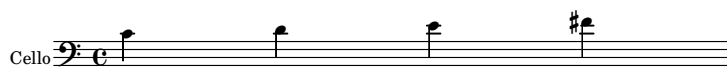
5.1.14 instrumenttools.Cello



class instrumenttools.**Cello** (*instrument_name='cello', short_instrument_name='vc.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('bass', 'tenor', 'treble'), pitch_range='[C2, G5]', sounding_pitch_of_written_middle_c=None*)

A cello.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> cello = instrumenttools.Cello()
>>> attach(cello, staff)
>>> show(staff)
```



Bases

- instrumenttools.Instrument
- abctools.AbjadObject
- __builtin__.object

Read-only properties

Cello.allowable_clefs

Gets cello's allowable clefs.

```
>>> cello.allowable_clefs
ClefInventory([Clef('bass'), Clef('tenor'), Clef('treble')])
```

```
>>> show(cello.allowable_clefs)
```



Returns clef inventory.

Cello.instrument_name

Gets cello's name.

```
>>> cello.instrument_name  
'cello'
```

Returns string.

Cello.instrument_name_markup

Gets cello's instrument name markup.

```
>>> cello.instrument_name_markup  
Markup(('Cello',))
```

```
>>> show(cello.instrument_name_markup)
```

Cello

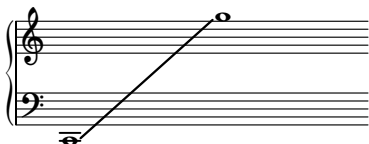
Returns markup.

Cello.pitch_range

Gets cello's range.

```
>>> cello.pitch_range  
PitchRange(' [C2, G5]')
```

```
>>> show(cello.pitch_range)
```



Returns pitch range.

Cello.short_instrument_name

Gets cello's short instrument name.

```
>>> cello.short_instrument_name  
'vc.'
```

Returns string.

Cello.short_instrument_name_markup

Gets cello's short instrument name markup.

```
>>> cello.short_instrument_name_markup  
Markup(('Vc.',))
```

```
>>> show(cello.short_instrument_name_markup)
```

Vc.

Returns markup.

Cello.sounding_pitch_of_written_middle_c

Gets sounding pitch of cello's written middle C.

```
>>> cello.sounding_pitch_of_written_middle_c  
NamedPitch('c')
```

```
>>> show(cello.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__ (*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__ (arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__ (format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__ ()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__ (instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__ (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

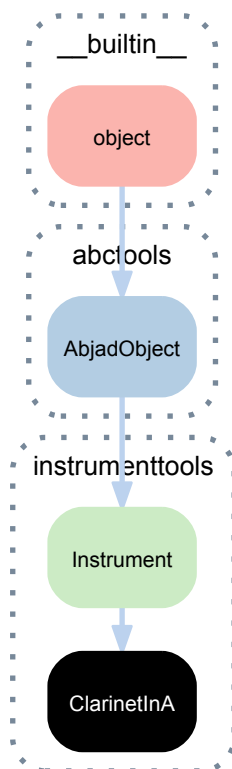
Returns boolean.

(Instrument) .__repr__ ()

Gets interpreter representation of instrument.

Returns string.

5.1.15 instrumenttools.ClarinetInA



```

class instrumenttools.ClarinetInA (instrument_name='clarinet' in A',
                                   short_instrument_name='cl. A \nat-
                                   ural', instrument_name_markup=None,
                                   short_instrument_name_markup=None, allow-
                                   able_clefs=None, pitch_range='[Db3, A6]', sound-
                                   ing_pitch_of_written_middle_c='A3')

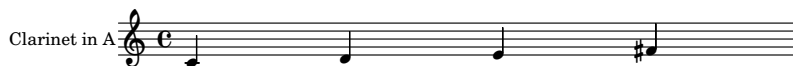
```

A clarinet in A.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clarinet = instrumenttools.ClarinetInA()
>>> attach(clarinet, staff)
>>> show(staff)

```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ClarinetInA.allowable_clefs`

Gets clarinet in A's allowable clefs.

```

>>> clarinet.allowable_clefs
ClefInventory([Clef('treble')])

```

```
>>> show(clarinet.allowable_clefs)
```



Returns clef inventory.

ClarinetInA.instrument_name

Gets clarinet in A's name.

```
>>> clarinet.instrument_name
'clarinet in A'
```

Returns string.

ClarinetInA.instrument_name_markup

Gets clarinet in A's instrument name markup.

```
>>> clarinet.instrument_name_markup
Markup(('Clarinet in A',))
```

```
>>> show(clarinet.instrument_name_markup)
```

Clarinet in A

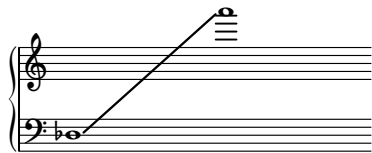
Returns markup.

ClarinetInA.pitch_range

Gets clarinet in A's range.

```
>>> clarinet.pitch_range
PitchRange(' [Db3, A6]')
```

```
>>> show(clarinet.pitch_range)
```



Returns pitch range.

ClarinetInA.short_instrument_name

Gets clarinet in A's short instrument name.

```
>>> clarinet.short_instrument_name
'cl. A \natural'
```

Returns string.

ClarinetInA.short_instrument_name_markup

Gets clarinet in A's short instrument name markup.

```
>>> clarinet.short_instrument_name_markup
Markup(('Cl.', 'A', MarkupCommand('natural')))
```

```
>>> show(clarinet.short_instrument_name_markup)
```

Cl. A \natural

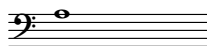
Returns markup.

ClarinetInA.sounding_pitch_of_written_middle_c

Gets sounding pitch of clarinet in A's written middle C.

```
>>> clarinet.sounding_pitch_of_written_middle_c
NamedPitch('a')
```

```
>>> show(clarinet.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__ (*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__ (arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__ (format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__ ()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__ (instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__ (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

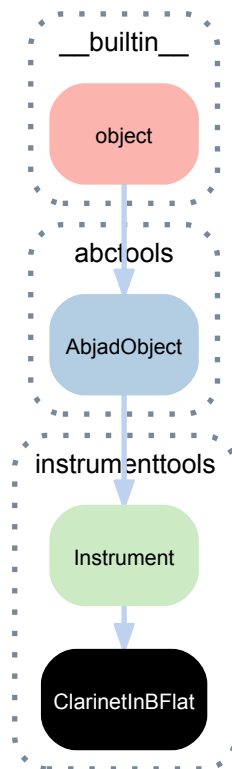
Returns boolean.

(Instrument) .__repr__ ()

Gets interpreter representation of instrument.

Returns string.

5.1.16 instrumenttools.ClarinetInBFlat



class instrumenttools.**ClarinetInBFlat** (*instrument_name='clarinet in B-flat', short_instrument_name='cl. in B-flat', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[D3, Bb6]', sounding_pitch_of_written_middle_c='Bb3'*)

A B-flat clarinet.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clarinet = instrumenttools.ClarinetInBFlat()
>>> attach(clarinet, staff)
>>> show(staff)
```



Bases

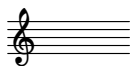
- instrumenttools.Instrument
- abctools.AbjadObject
- __builtin__.object

Read-only properties

ClarinetInBFlat.allowable_clefs
Gets clarinet in B-flat's allowable clefs.

```
>>> clarinet.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(clarinet.allowable_clefs)
```



Returns clef inventory.

`ClarinetInBFlat.instrument_name`

Gets clarinet in B-flat's name.

```
>>> clarinet.instrument_name
'clarinet in B-flat'
```

Returns string.

`ClarinetInBFlat.instrument_name_markup`

Gets clarinet in B-flat's instrument name markup.

```
>>> clarinet.instrument_name_markup
Markup(('Clarinet in B-flat',))
```

```
>>> show(clarinet.instrument_name_markup)
```

Clarinet in B-flat

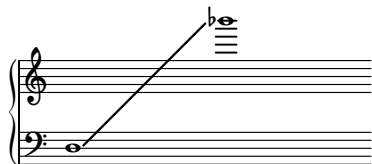
Returns markup.

`ClarinetInBFlat.pitch_range`

Gets clarinet in B-flat's range.

```
>>> clarinet.pitch_range
PitchRange('D3, Bb6')
```

```
>>> show(clarinet.pitch_range)
```



Returns pitch range.

`ClarinetInBFlat.short_instrument_name`

Gets clarinet in B-flat's short instrument name.

```
>>> clarinet.short_instrument_name
'cl. in B-flat'
```

Returns string.

`ClarinetInBFlat.short_instrument_name_markup`

Gets clarinet in B-flat's short instrument name markup.

```
>>> clarinet.short_instrument_name_markup
Markup(('Cl. in B-flat',))
```

```
>>> show(clarinet.short_instrument_name_markup)
```

Cl. in B-flat

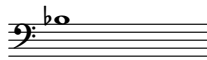
Returns markup.

`ClarinetInBFlat.sounding_pitch_of_written_middle_c`

Gets sounding pitch of clarinet in B-flat's written middle C.

```
>>> clarinet.sounding_pitch_of_written_middle_c
NamedPitch('bf')
```

```
>>> show(clarinet.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument).**__copy__**(*args)

Copies instrument.

Returns new instrument.

(Instrument).**__eq__**(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument).**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument).**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument).**__makenew__**(*instrument_name=None*, *short_instrument_name=None*, *instrument_name_markup=None*, *short_instrument_name_markup=None*, *allowable_clefs=None*, *pitch_range=None*, *sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

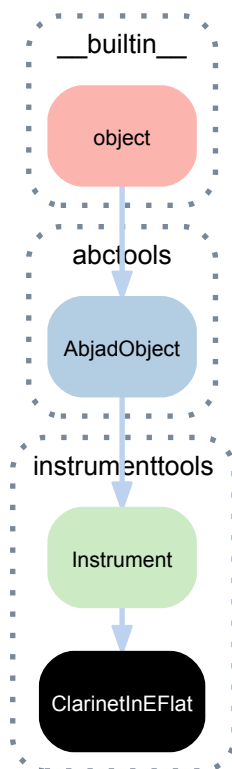
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.17 instrumenttools.ClarinetInEFlat



```
class instrumenttools.ClarinetInEFlat (instrument_name='clarinet' in E-
                                     flat', short_instrument_name='cl. E-
                                     flat', instrument_name_markup=None,
                                     short_instrument_name_markup=None, allow-
                                     able_clefs=None, pitch_range='[F3, C7]', sound-
                                     ing_pitch_of_written_middle_c='Eb4')
```

A E-flat clarinet.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clarinet = instrumenttools.ClarinetInEFlat()
>>> attach(clarinet, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ClarinetInEFlat.allowable_clefs`
Gets clarinet in E-flat's allowable clefs.

```
>>> clarinet.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(clarinet.allowable_clefs)
```



Returns clef inventory.

ClarinetInEFlat.instrument_name

Gets clarinet in E-flat's name.

```
>>> clarinet.instrument_name
'clarinet in E-flat'
```

Returns string.

ClarinetInEFlat.instrument_name_markup

Gets clarinet in E-flat's instrument name markup.

```
>>> clarinet.instrument_name_markup
Markup(('Clarinet in E-flat',))
```

```
>>> show(clarinet.instrument_name_markup)
```

Clarinet in E-flat

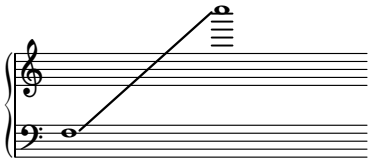
Returns markup.

ClarinetInEFlat.pitch_range

Gets clarinet in E-flat's range.

```
>>> clarinet.pitch_range
PitchRange(' [F3, C7]')
```

```
>>> show(clarinet.pitch_range)
```



Returns pitch range.

ClarinetInEFlat.short_instrument_name

Gets clarinet in E-flat's short instrument name.

```
>>> clarinet.short_instrument_name
'cl. E-flat'
```

Returns string.

ClarinetInEFlat.short_instrument_name_markup

Gets clarinet in E-flat's short instrument name markup.

```
>>> clarinet.short_instrument_name_markup
Markup(('Cl. E-flat',))
```

```
>>> show(clarinet.short_instrument_name_markup)
```

Cl. E-flat

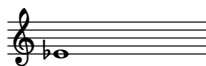
Returns markup.

ClarinetInEFlat.sounding_pitch_of_written_middle_c

Gets sounding pitch of clarinet in E-flat's written middle C.

```
>>> clarinet.sounding_pitch_of_written_middle_c
NamedPitch('ef')
```

```
>>> show(clarinet.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

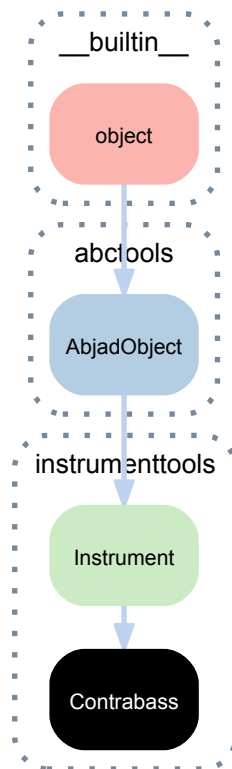
Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.18 instrumenttools.Contrabass



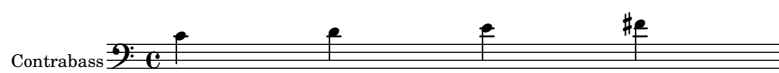
```

class instrumenttools.Contrabass (instrument_name='contrabass',
                                  short_instrument_name='vb.',
                                  instrument_name_markup=None,
                                  short_instrument_name_markup=None,
                                  allowable_clefs=('bass', 'treble'),
                                  pitch_range='[A1, D4]',
                                  sounding_pitch_of_written_middle_c='C3')
  
```

A contrabass.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> contrabass = instrumenttools.Contrabass()
>>> attach(contrabass, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`Contrabass.allowable_clefs`
Gets contrabass's allowable clefs.

```
>>> contrabass.allowable_clefs
ClefInventory([Clef('bass'), Clef('treble')])
```

```
>>> show(contrabass.allowable_clefs)
```



Returns clef inventory.

Contrabass.instrument_name

Gets contrabass's name.

```
>>> contrabass.instrument_name
'contrabass'
```

Returns string.

Contrabass.instrument_name_markup

Gets contrabass's instrument name markup.

```
>>> contrabass.instrument_name_markup
Markup(('Contrabass',))
```

```
>>> show(contrabass.instrument_name_markup)
```

Contrabass

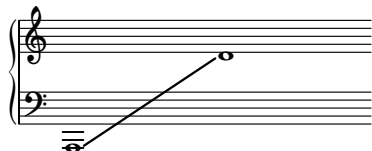
Returns markup.

Contrabass.pitch_range

Gets contrabass's range.

```
>>> contrabass.pitch_range
PitchRange('[A1, D4]')
```

```
>>> show(contrabass.pitch_range)
```



Returns pitch range.

Contrabass.short_instrument_name

Gets contrabass's short instrument name.

```
>>> contrabass.short_instrument_name
'vb.'
```

Returns string.

Contrabass.short_instrument_name_markup

Gets contrabass's short instrument name markup.

```
>>> contrabass.short_instrument_name_markup
Markup(('Vb.',))
```

```
>>> show(contrabass.short_instrument_name_markup)
```

Vb.

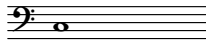
Returns markup.

Contrabass.sounding_pitch_of_written_middle_c

Gets sounding pitch of contrabass's written middle C.


```
>>> contrabass.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(contrabass.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument).**__copy__**(*args)

Copies instrument.

Returns new instrument.

(Instrument).**__eq__**(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument).**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument).**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument).**__makenew__**(*instrument_name=None*, *short_instrument_name=None*, *instrument_name_markup=None*, *short_instrument_name_markup=None*, *allowable_clefs=None*, *pitch_range=None*, *sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

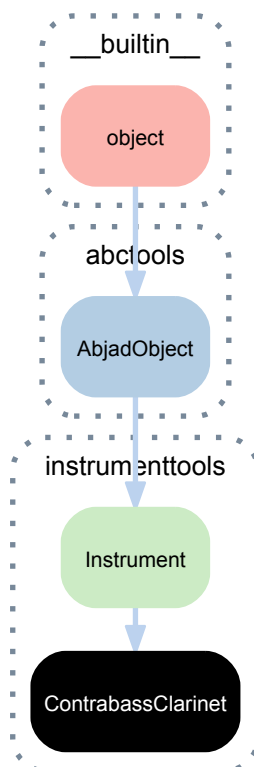
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.19 instrumenttools.ContrabassClarinet



```

class instrumenttools.ContrabassClarinet (instrument_name='contrabass      clar-
                                         inet',          short_instrument_name='cbass.
                                         cl.',          instrument_name_markup=None,
                                         short_instrument_name_markup=None,
                                         allowable_clefs=('treble',          'bass'),
                                         pitch_range='[Bb0,          G4]',          sound-
                                         ing_pitch_of_written_middle_c='Bb1')
  
```

A contrabass clarinet.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> contrabass_clarinet = instrumenttools.ContrabassClarinet()
>>> attach(contrabass_clarinet, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ContrabassClarinet.allowable_clefs`

Gets contrabass clarinet's allowable clefs.

```

>>> contrabass_clarinet.allowable_clefs
ClefInventory([Clef('treble'), Clef('bass')])
  
```

```
>>> show(contrabass_clarinet.allowable_clefs)
```



Returns clef inventory.

`ContrabassClarinet.instrument_name`

Gets contrabass clarinet's name.

```
>>> contrabass_clarinet.instrument_name
'contrabass clarinet'
```

Returns string.

`ContrabassClarinet.instrument_name_markup`

Gets contrabass clarinet's instrument name markup.

```
>>> contrabass_clarinet.instrument_name_markup
Markup(('Contrabass clarinet',))
```

```
>>> show(contrabass_clarinet.instrument_name_markup)
```

Contrabass clarinet

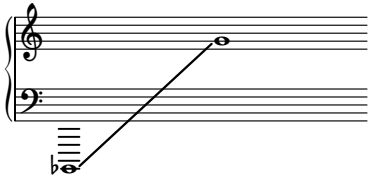
Returns markup.

`ContrabassClarinet.pitch_range`

Gets contrabass clarinet's range.

```
>>> contrabass_clarinet.pitch_range
PitchRange(' [Bb0, G4]')
```

```
>>> show(contrabass_clarinet.pitch_range)
```



Returns pitch range.

`ContrabassClarinet.short_instrument_name`

Gets contrabass clarinet's short instrument name.

```
>>> contrabass_clarinet.short_instrument_name
'cbass. cl.'
```

Returns string.

`ContrabassClarinet.short_instrument_name_markup`

Gets contrabass clarinet's short instrument name markup.

```
>>> contrabass_clarinet.short_instrument_name_markup
Markup(('Cbass. cl.',))
```

```
>>> show(contrabass_clarinet.short_instrument_name_markup)
```

Cbass. cl.

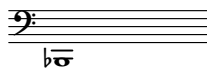
Returns markup.

`ContrabassClarinet.sounding_pitch_of_written_middle_c`

Gets sounding pitch of contrabass_clarinet's written middle C.

```
>>> contrabass_clarinet.sounding_pitch_of_written_middle_c
NamedPitch('bf,,')
```

```
>>> show(contrabass_clarinet.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__(*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__(format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

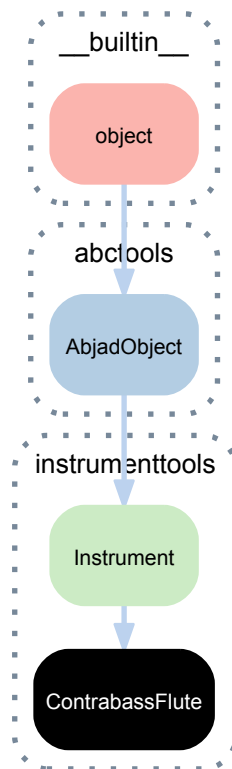
Returns boolean.

(Instrument) .__repr__()

Gets interpreter representation of instrument.

Returns string.

5.1.20 instrumenttools.ContrabassFlute



```

class instrumenttools.ContrabassFlute (instrument_name='contrabass      flute',
                                       short_instrument_name='cbass.      fl.',
                                       instrument_name_markup=None,
                                       short_instrument_name_markup=None,    allow-
                                       able_clefs=None, pitch_range='[G2, G5]', sound-
                                       ing_pitch_of_written_middle_c='G2')
  
```

A contrabass flute.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> contrabass_flute = instrumenttools.ContrabassFlute()
>>> attach(contrabass_flute, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ContrabassFlute.allowable_clefs`
Gets contrabass flute's allowable clefs.

```

>>> contrabass_flute.allowable_clefs
ClefInventory([Clef('treble')])
  
```

```
>>> show(contrabass_flute.allowable_clefs)
```



Returns clef inventory.

ContrabassFlute.instrument_name

Gets contrabass flute's name.

```
>>> contrabass_flute.instrument_name
'contrabass flute'
```

Returns string.

ContrabassFlute.instrument_name_markup

Gets contrabass flute's instrument name markup.

```
>>> contrabass_flute.instrument_name_markup
Markup(('Contrabass flute',))
```

```
>>> show(contrabass_flute.instrument_name_markup)
```

Contrabass flute

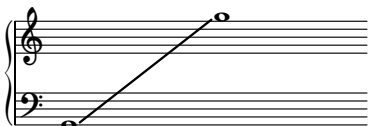
Returns markup.

ContrabassFlute.pitch_range

Gets contrabass flute's range.

```
>>> contrabass_flute.pitch_range
PitchRange(' [G2, G5]')
```

```
>>> show(contrabass_flute.pitch_range)
```



Returns pitch range.

ContrabassFlute.short_instrument_name

Gets contrabass flute's short instrument name.

```
>>> contrabass_flute.short_instrument_name
'cbass. fl.'
```

Returns string.

ContrabassFlute.short_instrument_name_markup

Gets contrabass flute's short instrument name markup.

```
>>> contrabass_flute.short_instrument_name_markup
Markup(('Cbass. fl.',))
```

```
>>> show(contrabass_flute.short_instrument_name_markup)
```

Cbass. fl.

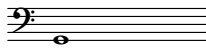
Returns markup.

ContrabassFlute.sounding_pitch_of_written_middle_c

Gets sounding pitch of contrabass_flute's written middle C.

```
>>> contrabass_flute.sounding_pitch_of_written_middle_c
NamedPitch('g,')
```

```
>>> show(contrabass_flute.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__** (*args)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__** (arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__** (format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__** ()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__** (instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__** (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

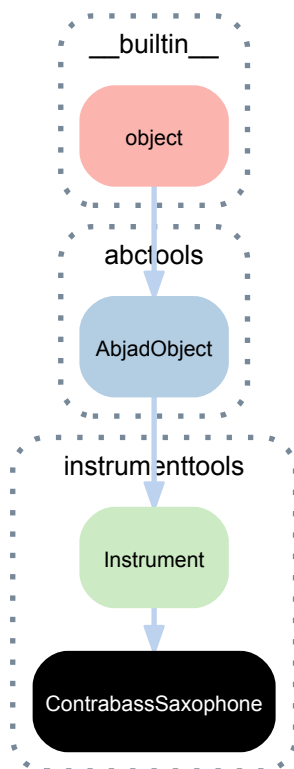
Returns boolean.

(Instrument) .**__repr__** ()

Gets interpreter representation of instrument.

Returns string.

5.1.21 instrumenttools.ContrabassSaxophone



class instrumenttools.ContrabassSaxophone (*instrument_name='contrabass saxophone', short_instrument_name='cbass. sax.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[C1, Ab3]', sounding_pitch_of_written_middle_c='Eb1'*)

A bass saxophone.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> contrabass_saxophone = instrumenttools.ContrabassSaxophone()
>>> attach(contrabass_saxophone, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

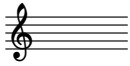
ContrabassSaxophone.allowable_clefs
Gets contrabass saxophone's allowable clefs.

```

>>> contrabass_saxophone.allowable_clefs
ClefInventory([Clef('treble')])
  
```



```
>>> show(contrabass_saxophone.allowable_clefs)
```



Returns clef inventory.

`ContrabassSaxophone.instrument_name`

Gets contrabass saxophone's name.

```
>>> contrabass_saxophone.instrument_name
'contrabass saxophone'
```

Returns string.

`ContrabassSaxophone.instrument_name_markup`

Gets contrabass saxophone's instrument name markup.

```
>>> contrabass_saxophone.instrument_name_markup
Markup(('Contrabass saxophone',))
```

```
>>> show(contrabass_saxophone.instrument_name_markup)
```

Contrabass saxophone

Returns markup.

`ContrabassSaxophone.pitch_range`

Gets contrabass saxophone's range.

```
>>> contrabass_saxophone.pitch_range
PitchRange('C1, Ab3')
```

```
>>> show(contrabass_saxophone.pitch_range)
```



Returns pitch range.

`ContrabassSaxophone.short_instrument_name`

Gets contrabass saxophone's short instrument name.

```
>>> contrabass_saxophone.short_instrument_name
'cbass. sax.'
```

Returns string.

`ContrabassSaxophone.short_instrument_name_markup`

Gets contrabass saxophone's short instrument name markup.

```
>>> contrabass_saxophone.short_instrument_name_markup
Markup(('Cbass. sax.',))
```

```
>>> show(contrabass_saxophone.short_instrument_name_markup)
```

Cbass. sax.

Returns markup.

`ContrabassSaxophone.sounding_pitch_of_written_middle_c`

Gets sounding pitch of contrabass_saxophone's written middle C.

```
>>> contrabass_saxophone.sounding_pitch_of_written_middle_c
NamedPitch('ef,,')
```

```
>>> show(contrabass_saxophone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__ (*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__ (arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__ (format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__ ()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__ (instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__ (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

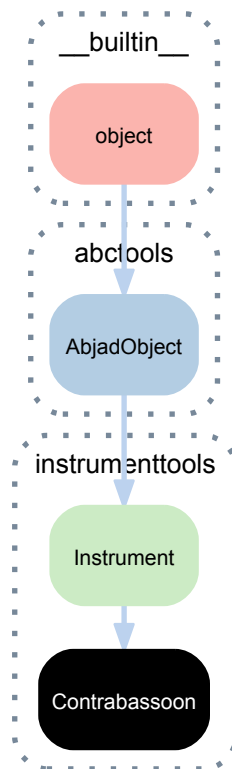
Returns boolean.

(Instrument) .__repr__ ()

Gets interpreter representation of instrument.

Returns string.

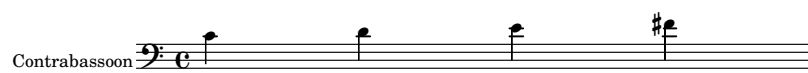
5.1.22 instrumenttools.Contrabassoon



```
class instrumenttools.Contrabassoon(instrument_name='contrabassoon',
                                   short_instrument_name='contrabsn.',
                                   instrument_name_markup=None,
                                   short_instrument_name_markup=None,      allow-
                                   able_clefs=('bass', ), pitch_range='[Bb0, Bb4]',
                                   sounding_pitch_of_written_middle_c='C3')
```

A contrabassoon.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> contrabassoon = instrumenttools.Contrabassoon()
>>> attach(contrabassoon, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`Contrabassoon.allowable_clefs`
Gets contrabassoon's allowable clefs.

```
>>> contrabassoon.allowable_clefs
ClefInventory([Clef('bass')])
```

```
>>> show(contrabassoon.allowable_clefs)
```



Returns clef inventory.

Contrabassoon.instrument_name

Gets contrabassoon's name.

```
>>> contrabassoon.instrument_name
'contrabassoon'
```

Returns string.

Contrabassoon.instrument_name_markup

Gets contrabassoon's instrument name markup.

```
>>> contrabassoon.instrument_name_markup
Markup(('Contrabassoon',))
```

```
>>> show(contrabassoon.instrument_name_markup)
```

Contrabassoon

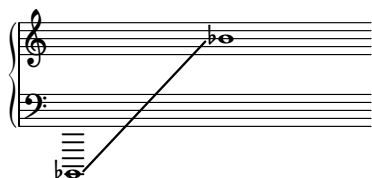
Returns markup.

Contrabassoon.pitch_range

Gets contrabassoon's range.

```
>>> contrabassoon.pitch_range
PitchRange(' [Bb0, Bb4]')
```

```
>>> show(contrabassoon.pitch_range)
```



Returns pitch range.

Contrabassoon.short_instrument_name

Gets contrabassoon's short instrument name.

```
>>> contrabassoon.short_instrument_name
'contrabsn.'
```

Returns string.

Contrabassoon.short_instrument_name_markup

Gets contrabassoon's short instrument name markup.

```
>>> contrabassoon.short_instrument_name_markup
Markup(('Contrabsn.',))
```

```
>>> show(contrabassoon.short_instrument_name_markup)
```

Contrabsn.

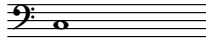
Returns markup.

Contrabassoon.sounding_pitch_of_written_middle_c

Gets sounding pitch of contrabassoon's written middle C.

```
>>> contrabassoon.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(contrabassoon.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

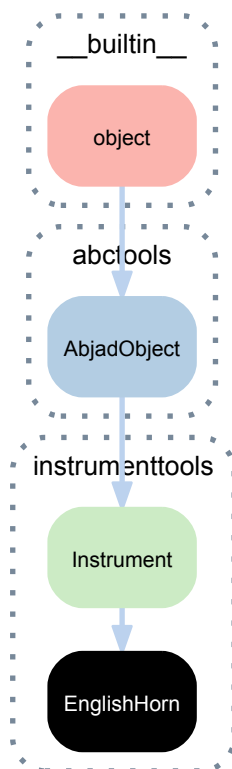
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.23 instrumenttools.EnglishHorn



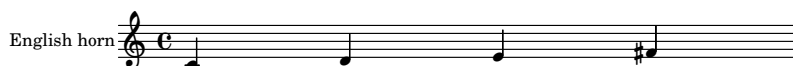
```

class instrumenttools.EnglishHorn (instrument_name='English horn',
                                   short_instrument_name='Eng. hn.',
                                   instrument_name_markup=None,
                                   short_instrument_name_markup=None,
                                   allowable_clefs=None, pitch_range='[E3, C6]', allow-
                                   ing_pitch_of_written_middle_c='F3') sound-
  
```

A English horn.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> english_horn = instrumenttools.EnglishHorn()
>>> attach(english_horn, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

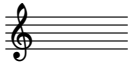
`EnglishHorn.allowable_clefs`

Gets English horn's allowable clefs.

```

>>> english_horn.allowable_clefs
ClefInventory([Clef('treble')])
  
```

```
>>> show(english_horn.allowable_clefs)
```



Returns clef inventory.

EnglishHorn.instrument_name
Gets English horn's name.

```
>>> english_horn.instrument_name
'English horn'
```

Returns string.

EnglishHorn.instrument_name_markup
Gets English horn's instrument name markup.

```
>>> english_horn.instrument_name_markup
Markup(('English horn',))
```

```
>>> show(english_horn.instrument_name_markup)
```

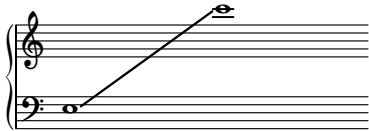
English horn

Returns markup.

EnglishHorn.pitch_range
Gets English horn's range.

```
>>> english_horn.pitch_range
PitchRange('E3, C6')
```

```
>>> show(english_horn.pitch_range)
```



Returns pitch range.

EnglishHorn.short_instrument_name
Gets English horn's short instrument name.

```
>>> english_horn.short_instrument_name
'Eng. hn.'
```

Returns string.

EnglishHorn.short_instrument_name_markup
Gets English horn's short instrument name markup.

```
>>> english_horn.short_instrument_name_markup
Markup(('Eng. hn.',))
```

```
>>> show(english_horn.short_instrument_name_markup)
```

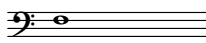
Eng. hn.

Returns markup.

EnglishHorn.sounding_pitch_of_written_middle_c
Gets sounding pitch of English horn's written middle C.

```
>>> english_horn.sounding_pitch_of_written_middle_c
NamedPitch('f')
```

```
>>> show(english_horn.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

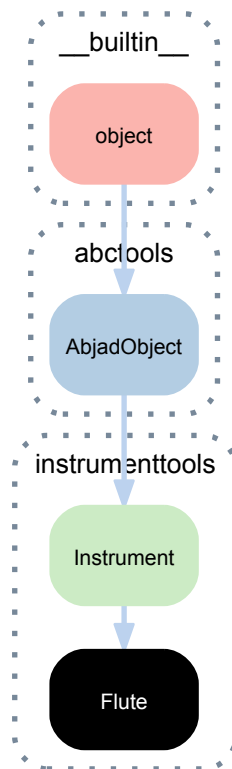
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

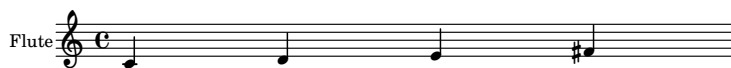
5.1.24 instrumenttools.Flute



class instrumenttools.**Flute**(*instrument_name='flute', short_instrument_name='fl.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[C4, D7]', sounding_pitch_of_written_middle_c=None*)

A flute.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> flute = instrumenttools.Flute()
>>> attach(flute, staff)
>>> show(staff)
```



Bases

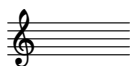
- instrumenttools.Instrument
- abctools.AbjadObject
- __builtin__.object

Read-only properties

Flute.allowable_clefs
Gets flute's allowable clefs.

```
>>> flute.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(flute.allowable_clefs)
```



Returns clef inventory.

Flute.instrument_name
Gets flute's name.

```
>>> flute.instrument_name
'flute'
```

Returns string.

Flute.instrument_name_markup
Gets flute's instrument name markup.

```
>>> flute.instrument_name_markup
Markup(('Flute',))
```

```
>>> show(flute.instrument_name_markup)
```

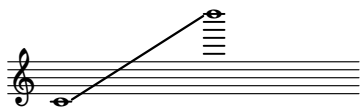
Flute

Returns markup.

Flute.pitch_range
Gets flute's range.

```
>>> flute.pitch_range
PitchRange(' [C4, D7]')
```

```
>>> show(flute.pitch_range)
```



Returns pitch range.

Flute.short_instrument_name
Gets flute's short instrument name.

```
>>> flute.short_instrument_name
'fl.'
```

Returns string.

Flute.short_instrument_name_markup
Gets flute's short instrument name markup.

```
>>> flute.short_instrument_name_markup
Markup(('Fl.',))
```

```
>>> show(flute.short_instrument_name_markup)
```

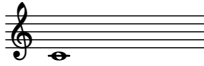
Fl.

Returns markup.

Flute.sounding_pitch_of_written_middle_c
Gets sounding pitch of flute's written middle C.

```
>>> flute.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(flute.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

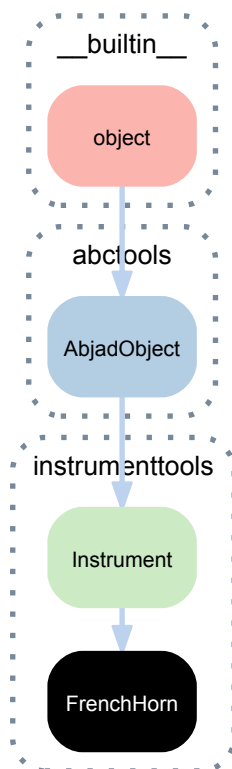
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

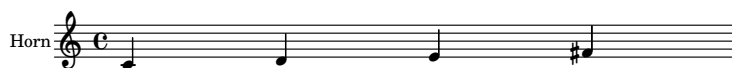
5.1.25 instrumenttools.FrenchHorn



```
class instrumenttools.FrenchHorn (instrument_name='horn',    short_instrument_name='hn.',
                                instrument_name_markup=None,
                                short_instrument_name_markup=None,    allowable_clefs=('bass', 'treble'), pitch_range='[B1, F5]',
                                sounding_pitch_of_written_middle_c='F3')
```

A French horn.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> french_horn = instrumenttools.FrenchHorn()
>>> attach(french_horn, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`FrenchHorn.allowable_clefs`

Gets French horn's allowable clefs.

```
>>> french_horn.allowable_clefs
ClefInventory([Clef('bass'), Clef('treble')])
```

```
>>> show(french_horn.allowable_clefs)
```



Returns clef inventory.

FrenchHorn.instrument_name
Gets French horn's name.

```
>>> french_horn.instrument_name
'horn'
```

Returns string.

FrenchHorn.instrument_name_markup
Gets French horn's instrument name markup.

```
>>> french_horn.instrument_name_markup
Markup(('Horn',))
```

```
>>> show(french_horn.instrument_name_markup)
```

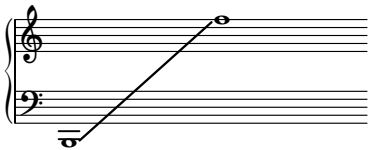
Horn

Returns markup.

FrenchHorn.pitch_range
Gets French horn's range.

```
>>> french_horn.pitch_range
PitchRange('B1, F5')
```

```
>>> show(french_horn.pitch_range)
```



Returns pitch range.

FrenchHorn.short_instrument_name
Gets French horn's short instrument name.

```
>>> french_horn.short_instrument_name
'hn.'
```

Returns string.

FrenchHorn.short_instrument_name_markup
Gets French horn's short instrument name markup.

```
>>> french_horn.short_instrument_name_markup
Markup(('Hn.',))
```

```
>>> show(french_horn.short_instrument_name_markup)
```

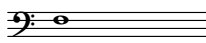
Hn.

Returns markup.

FrenchHorn.sounding_pitch_of_written_middle_c
Gets sounding pitch of French horn's written middle C.

```
>>> french_horn.sounding_pitch_of_written_middle_c
NamedPitch('f')
```

```
>>> show(french_horn.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

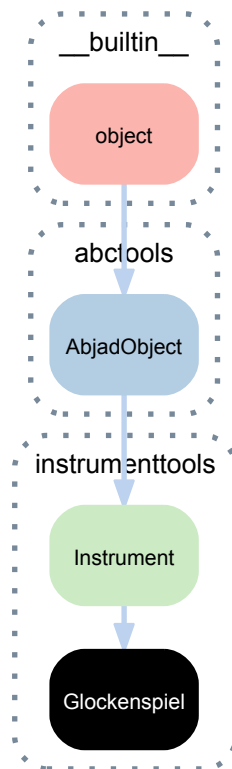
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

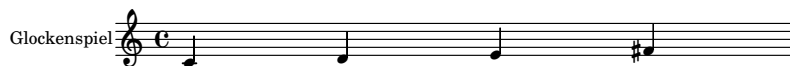
5.1.26 instrumenttools.Glockenspiel



```
class instrumenttools.Glockenspiel (instrument_name='glockenspiel',
                                     short_instrument_name='gkspl.',          in-
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,        allow-
                                     allowable_clefs=None, pitch_range='[G5, C8]', sound-
                                     ing_pitch_of_written_middle_c='C6')
```

A glockenspiel.

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



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

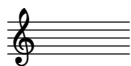
Read-only properties

`Glockenspiel.allowable_clefs`

Gets glockenspiel's allowable clefs.

```
>>> glockenspiel.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(glockenspiel.allowable_clefs)
```



Returns clef inventory.

Glockenspiel.instrument_name

Gets glockenspiel's name.

```
>>> glockenspiel.instrument_name
'glockenspiel'
```

Returns string.

Glockenspiel.instrument_name_markup

Gets glockenspiel's instrument name markup.

```
>>> glockenspiel.instrument_name_markup
Markup(('Glockenspiel',))
```

```
>>> show(glockenspiel.instrument_name_markup)
```

Glockenspiel

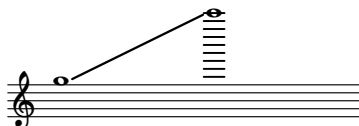
Returns markup.

Glockenspiel.pitch_range

Gets glockenspiel's range.

```
>>> glockenspiel.pitch_range
PitchRange('[G5, C8]')
```

```
>>> show(glockenspiel.pitch_range)
```



Returns pitch range.

Glockenspiel.short_instrument_name

Gets glockenspiel's short instrument name.

```
>>> glockenspiel.short_instrument_name
'gkspl.'
```

Returns string.

Glockenspiel.short_instrument_name_markup

Gets glockenspiel's short instrument name markup.

```
>>> glockenspiel.short_instrument_name_markup
Markup(('Gkspl.',))
```

```
>>> show(glockenspiel.short_instrument_name_markup)
```

Gkspl.

Returns markup.

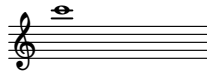
Glockenspiel.sounding_pitch_of_written_middle_c

Gets sounding pitch of glockenspiel's written middle C.

```
>>> glockenspiel.sounding_pitch_of_written_middle_c
NamedPitch('c''')
```



```
>>> show(glockenspiel.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

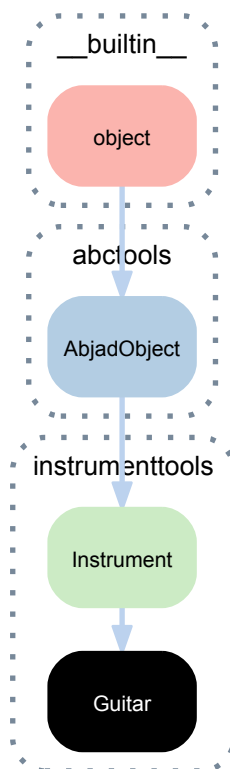
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

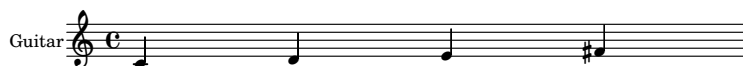
5.1.27 instrumenttools.Guitar



class instrumenttools.Guitar (*instrument_name='guitar', short_instrument_name='gt.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[E2, E5]', sounding_pitch_of_written_middle_c='C3'*)

A guitar.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> guitar = instrumenttools.Guitar()
>>> attach(guitar, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

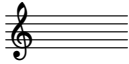
Read-only properties

Guitar.allowable_clefs

Gets guitar's allowable clefs.

```
>>> guitar.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(guitar.allowable_clefs)
```



Returns clef inventory.

Guitar.**instrument_name**
Gets guitar's name.

```
>>> guitar.instrument_name
'guitar'
```

Returns string.

Guitar.**instrument_name_markup**
Gets guitar's instrument name markup.

```
>>> guitar.instrument_name_markup
Markup(('Guitar',))
```

```
>>> show(guitar.instrument_name_markup)
```

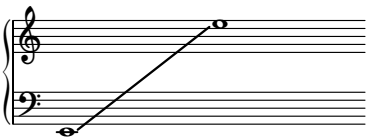
Guitar

Returns markup.

Guitar.**pitch_range**
Gets guitar's range.

```
>>> guitar.pitch_range
PitchRange('[E2, E5]')
```

```
>>> show(guitar.pitch_range)
```



Returns pitch range.

Guitar.**short_instrument_name**
Gets guitar's short instrument name.

```
>>> guitar.short_instrument_name
'gt.'
```

Returns string.

Guitar.**short_instrument_name_markup**
Gets guitar's short instrument name markup.

```
>>> guitar.short_instrument_name_markup
Markup(('Gt.',))
```

```
>>> show(guitar.short_instrument_name_markup)
```

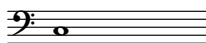
Gt.

Returns markup.

Guitar.**sounding_pitch_of_written_middle_c**
Gets sounding pitch of guitar's written middle C.

```
>>> guitar.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(guitar.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

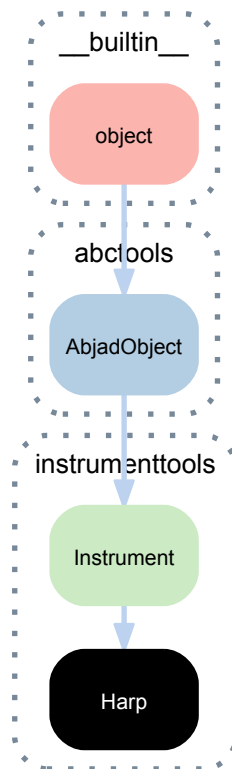
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.28 instrumenttools.Harp



class `instrumenttools.Harp` (*instrument_name='harp', short_instrument_name='hp.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('treble', 'bass'), pitch_range='[B0, G#7]', sounding_pitch_of_written_middle_c=None*)

A harp.

```

>>> piano_staff = scoretools.PianoStaff()
>>> piano_staff.append(Staff("c'4 d'4 e'4 f'4"))
>>> piano_staff.append(Staff("c'2 b2"))
>>> harp = instrumenttools.Harp()
>>> attach(harp, piano_staff)
>>> attach(Clef('bass'), piano_staff[1])
>>> show(piano_staff)
  
```



The harp targets piano staff context by default.

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`Harp.allowable_clefs`
Gets harp's allowable clefs.

```
>>> harp.allowable_clefs
ClefInventory([Clef('treble'), Clef('bass')])
```

```
>>> show(harp.allowable_clefs)
```



Returns clef inventory.

Harp.**instrument_name**

Gets harp's name.

```
>>> harp.instrument_name
'harp'
```

Returns string.

Harp.**instrument_name_markup**

Gets harp's instrument name markup.

```
>>> harp.instrument_name_markup
Markup(('Harp',))
```

```
>>> show(harp.instrument_name_markup)
```

Harp

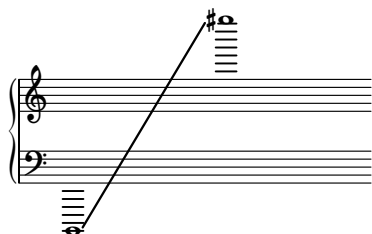
Returns markup.

Harp.**pitch_range**

Gets harp's range.

```
>>> harp.pitch_range
PitchRange(' [B0, G#7]')
```

```
>>> show(harp.pitch_range)
```



Returns pitch range.

Harp.**short_instrument_name**

Gets harp's short instrument name.

```
>>> harp.short_instrument_name
'hp.'
```

Returns string.

Harp.**short_instrument_name_markup**

Gets harp's short instrument name markup.

```
>>> harp.short_instrument_name_markup
Markup(('Hp.',))
```

```
>>> show(harp.short_instrument_name_markup)
```

Hp.

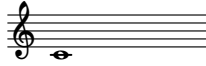
Returns markup.

`Harp.sounding_pitch_of_written_middle_c`

Gets sounding pitch of harp's written middle C.

```
>>> harp.sounding_pitch_of_written_middle_c
NamedPitch("c")
```

```
>>> show(harp.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

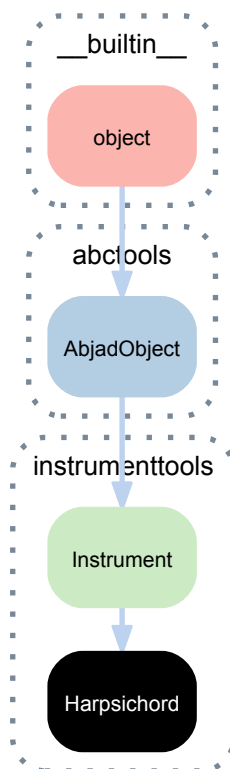
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.29 instrumenttools.Harpsichord



```

class instrumenttools.Harpsichord(instrument_name='harpsichord',
                                   short_instrument_name='hpschd.',
                                   instrument_name_markup=None,
                                   short_instrument_name_markup=None, allow-
                                   able_clefs=('treble', 'bass'), pitch_range='[C2, C7]',
                                   sounding_pitch_of_written_middle_c=None)
  
```

A harpsichord.

```

>>> upper_staff = Staff("c'4 d'4 e'4 f'4")
>>> lower_staff = Staff("c'2 b2")
>>> piano_staff = scoretools.PianoStaff([upper_staff, lower_staff])
>>> harpsichord = instrumenttools.Harpsichord()
>>> attach(harpsichord, piano_staff)
>>> attach(Clef('bass'), lower_staff)
>>> show(piano_staff)
  
```



The harpsichord targets piano staff context by default.

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

Harpsichord.allowable_clefs

Gets harpsichord's allowable clefs.

```
>>> harpsichord.allowable_clefs
ClefInventory([Clef('treble'), Clef('bass')])
```

```
>>> show(harpsichord.allowable_clefs)
```



Returns clef inventory.

Harpsichord.instrument_name

Gets harpsichord's name.

```
>>> harpsichord.instrument_name
'harpsichord'
```

Returns string.

Harpsichord.instrument_name_markup

Gets harpsichord's instrument name markup.

```
>>> harpsichord.instrument_name_markup
Markup(('Harpsichord',))
```

```
>>> show(harpsichord.instrument_name_markup)
```

Harpsichord

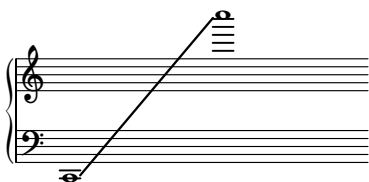
Returns markup.

Harpsichord.pitch_range

Gets harpsichord's range.

```
>>> harpsichord.pitch_range
PitchRange(' [C2, C7]')
```

```
>>> show(harpsichord.pitch_range)
```



Returns pitch range.

Harpsichord.short_instrument_name

Gets harpsichord's short instrument name.

```
>>> harpsichord.short_instrument_name
'hpschd.'
```

Returns string.

Harpsichord.short_instrument_name_markup

Gets harpsichord's short instrument name markup.

```
>>> harpsichord.short_instrument_name_markup
Markup(('Hpschd.',))
```

```
>>> show(harpsichord.short_instrument_name_markup)
```

Hpschd.

Returns markup.

Harpsichord.sounding_pitch_of_written_middle_c

Gets sounding pitch of harpsichord's written middle C.

```
>>> harpsichord.sounding_pitch_of_written_middle_c
NamedPitch("c' ")
```

```
>>> show(harpsichord.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

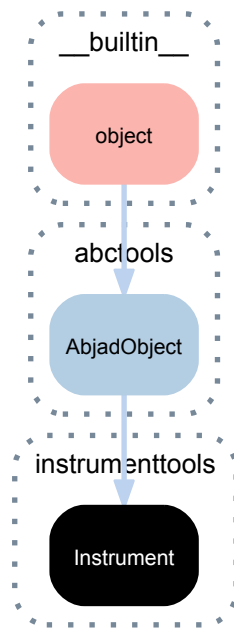
Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.30 instrumenttools.Instrument



```

class instrumenttools.Instrument (instrument_name=None,    short_instrument_name=None,
                                instrument_name_markup=None,
                                short_instrument_name_markup=None,    allow-
                                able_clefs=None,    pitch_range=None,    sound-
                                ing_pitch_of_written_middle_c=None)
  
```

A musical instrument.

Bases

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

Read-only properties

`Instrument.allowable_clefs`
 Gets allowable clefs of instrument.
 Returns clef inventory.

`Instrument.instrument_name`
 Gets instrument name.
 Returns string.

`Instrument.instrument_name_markup`
 Gets instrument name markup.
 Returns markup.

`Instrument.pitch_range`
 Gets pitch range of instrument.
 Returns pitch range.

`Instrument.short_instrument_name`
 Gets short instrument name.
 Returns string.

`Instrument.short_instrument_name_markup`

Gets short instrument name markup.

Returns markup.

`Instrument.sounding_pitch_of_written_middle_c`

Gets sounding pitch of written middle C.

Returns named pitch.

Special methods

`Instrument.__copy__(*args)`

Copies instrument.

Returns new instrument.

`Instrument.__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`Instrument.__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`Instrument.__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`Instrument.__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

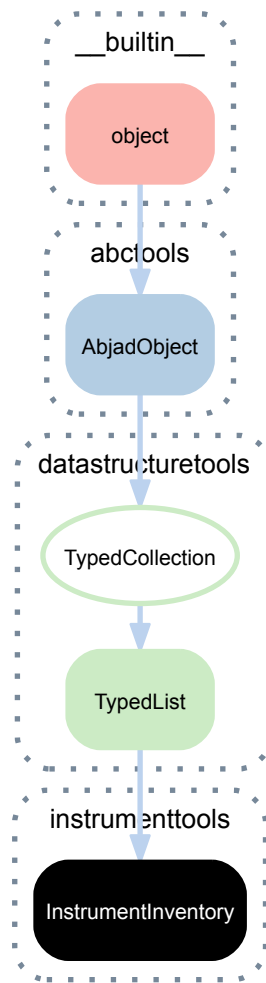
Returns boolean.

`Instrument.__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.31 instrumenttools.InstrumentInventory



class `instrumenttools.InstrumentInventory` (*tokens=None*, *item_class=None*, *keep_sorted=None*, *custom_identifier=None*)

An ordered list of instruments.

```
>>> inventory = instrumenttools.InstrumentInventory([
...     instrumenttools.Flute(),
...     instrumenttools.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.

Read/write properties

(TypedCollection) .**custom_identifier**
Gets and sets custom identifier of typed collection.

Returns string or none.

(TypedList) .**keep_sorted**
Sorts collection on mutation if true.

Methods

(TypedList) .**append** (*token*)
Changes *token* to item and appends.

```
>>> 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*)
Changes *token* to item and returns 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*)
Changes *tokens* to items and extends.

```
>>> 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*)
Changes *token* to item and returns index.

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

Returns index.

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

```
>>> 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.

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

Aliases list.pop().

(TypedList) **.remove** (*token*)

Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

InstrumentInventory **.__format__** (*format_specification=''*)

Formats instrument inventory.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList) **.__getitem__** (*i*)

Aliases list.__getitem__().

(TypedList) **.__iadd__** (*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList(
[
    indicatortools.Dynamic(
        'ppp'
    ),
    indicatortools.Dynamic(
        'p'
    ),
    indicatortools.Dynamic(
        'mp'
    ),
    indicatortools.Dynamic(
        'mf'
    ),
    indicatortools.Dynamic(
        'fff'
    ),
],
item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

InstrumentInventory.**__repr__**()

Gets interpreter representation of instrument inventory.

```
>>> inventory
InstrumentInventory([Flute(), Guitar()])
```

Returns string.

(TypedList).**__reversed__**()

Aliases list.**__reversed__**().

(TypedList).**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

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



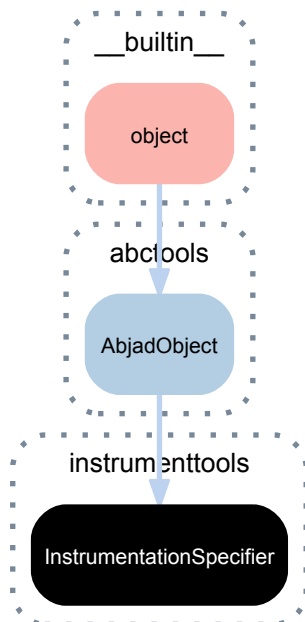
```

        pitchtools.NamedPitch('gqs, '),
    ],
    item_class=pitchtools.NamedPitch,
)

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print format(pitch_collection)
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,
)

```

5.1.32 instrumenttools.InstrumentationSpecifier



class `instrumenttools.InstrumentationSpecifier` (*performers=None*)
 Instrumentation specifier for an entire score.

```

>>> flute = instrumenttools.Performer('Flute')
>>> flute.instruments.append(instrumenttools.Flute())
>>> flute.instruments.append(instrumenttools.AltoFlute())

```

```

>>> guitar = instrumenttools.Performer('Guitar')
>>> guitar.instruments.append(instrumenttools.Guitar())

```

```

>>> instrumentation_specifier = \
...     instrumenttools.InstrumentationSpecifier([flute, guitar])

```

```

>>> print format(instrumentation_specifier)
instrumenttools.InstrumentationSpecifier(
    performers=instrumenttools.PerformerInventory (
        [
            instrumenttools.Performer (
                name='Flute',
                instruments=instrumenttools.InstrumentInventory (

```

```

[
    instrumenttools.Flute(
        instrument_name='flute',
        short_instrument_name='fl.',
        instrument_name_markup=markuptools.Markup(
            ('Flute',)
        ),
        short_instrument_name_markup=markuptools.Markup(
            ('Fl.',)
        ),
        allowable_clefs=indicatortools.ClefInventory(
            [
                indicatortools.Clef(
                    'treble'
                ),
            ]
        ),
        pitch_range=pitchtools.PitchRange(
            '[C4, D7]'
        ),
        sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('c'),
    ),
    instrumenttools.AltoFlute(
        instrument_name='alto flute',
        short_instrument_name='alt. fl.',
        instrument_name_markup=markuptools.Markup(
            ('Alto flute',)
        ),
        short_instrument_name_markup=markuptools.Markup(
            ('Alt. fl.',)
        ),
        allowable_clefs=indicatortools.ClefInventory(
            [
                indicatortools.Clef(
                    'treble'
                ),
            ]
        ),
        pitch_range=pitchtools.PitchRange(
            '[G3, G6]'
        ),
        sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('g'),
    ),
],
),
instrumenttools.Performer(
    name='Guitar',
    instruments=instrumenttools.InstrumentInventory(
        [
            instrumenttools.Guitar(
                instrument_name='guitar',
                short_instrument_name='gt.',
                instrument_name_markup=markuptools.Markup(
                    ('Guitar',)
                ),
                short_instrument_name_markup=markuptools.Markup(
                    ('Gt.',)
                ),
                allowable_clefs=indicatortools.ClefInventory(
                    [
                        indicatortools.Clef(
                            'treble'
                        ),
                    ]
                ),
                pitch_range=pitchtools.PitchRange(
                    '[E2, E5]'
                ),
                sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('c'),
            ),
        ]
    ),
),

```

```

        ),
    ],
),
)

```

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.

Read/write properties

`InstrumentationSpecifier.performers`
Gets and sets list of performers in score.

```
>>> print format(instrumentation_specifier.performers)
instrumenttools.PerformerInventory(
  [
    instrumenttools.Performer(
      name='Flute',
      instruments=instrumenttools.InstrumentInventory(
        [
          instrumenttools.Flute(
            instrument_name='flute',
            short_instrument_name='fl.',
            instrument_name_markup=markuptools.Markup(
              ('Flute',)
            ),

```

```

        short_instrument_name_markup=markuptools.Markup(
            ('Fl.',),
        ),
        allowable_clefs=indicatortools.ClefInventory(
            [
                indicatortools.Clef(
                    'treble'
                ),
            ]
        ),
        pitch_range=pitchtools.PitchRange(
            '[C4, D7]'
        ),
        sounding_pitch_of_written_middle_c=pitchtools.NamedPitch("c"),
    ),
    instrumenttools.AltoFlute(
        instrument_name='alto flute',
        short_instrument_name='alt. fl.',
        instrument_name_markup=markuptools.Markup(
            ('Alto flute',),
        ),
        short_instrument_name_markup=markuptools.Markup(
            ('Alt. fl.',),
        ),
        allowable_clefs=indicatortools.ClefInventory(
            [
                indicatortools.Clef(
                    'treble'
                ),
            ]
        ),
        pitch_range=pitchtools.PitchRange(
            '[G3, G6]'
        ),
        sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('g'),
    ),
    ],
),
instrumenttools.Performer(
    name='Guitar',
    instruments=instrumenttools.InstrumentInventory(
        [
            instrumenttools.Guitar(
                instrument_name='guitar',
                short_instrument_name='gt.',
                instrument_name_markup=markuptools.Markup(
                    ('Guitar',),
                ),
                short_instrument_name_markup=markuptools.Markup(
                    ('Gt.',),
                ),
                allowable_clefs=indicatortools.ClefInventory(
                    [
                        indicatortools.Clef(
                            'treble'
                        ),
                    ]
                ),
                pitch_range=pitchtools.PitchRange(
                    '[E2, E5]'
                ),
                sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('c'),
            ),
        ]
    ),
),
]
)

```

Returns performer inventory.

Special methods

`InstrumentationSpecifier.__eq__(expr)`

True when *expr* is an instrumentation specifier with performers equal to those of instrumentation specifier. Otherwise false.

Returns boolean.

`InstrumentationSpecifier.__format__(format_specification='')`

Formats instrumentation specifier.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

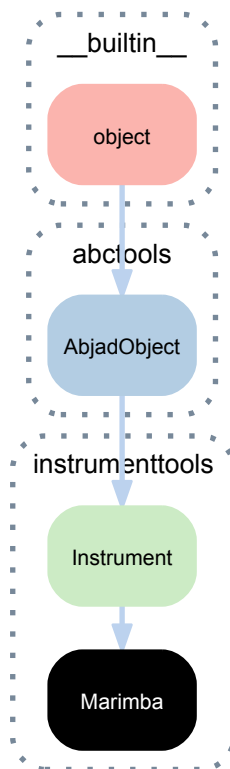
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

5.1.33 instrumenttools.Marimba

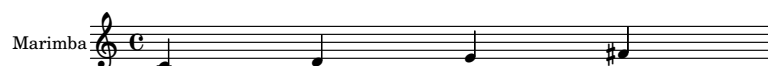


```
class instrumenttools.Marimba(instrument_name='marimba', short_instrument_name='mb.',
                              instrument_name_markup=None,
                              short_instrument_name_markup=None, allowable_clefs=('treble', 'bass'), pitch_range='[F2, C7]', sounding_pitch_of_written_middle_c=None)
```

A marimba.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> marimba = instrumenttools.Marimba()
```

```
>>> attach(marimba, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

Marimba.allowable_clefs
Gets marimba's allowable clefs.

```
>>> marimba.allowable_clefs
ClefInventory([Clef('treble'), Clef('bass')])
```

```
>>> show(marimba.allowable_clefs)
```



Returns clef inventory.

Marimba.instrument_name
Gets marimba's name.

```
>>> marimba.instrument_name
'marimba'
```

Returns string.

Marimba.instrument_name_markup
Gets marimba's instrument name markup.

```
>>> marimba.instrument_name_markup
Markup(('Marimba',))
```

```
>>> show(marimba.instrument_name_markup)
```

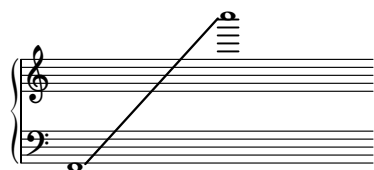
Marimba

Returns markup.

Marimba.pitch_range
Gets marimba's range.

```
>>> marimba.pitch_range
PitchRange('F2, C7')
```

```
>>> show(marimba.pitch_range)
```



Returns pitch range.

`Marimba.short_instrument_name`

Gets marimba's short instrument name.

```
>>> marimba.short_instrument_name
'mb.'
```

Returns string.

`Marimba.short_instrument_name_markup`

Gets marimba's short instrument name markup.

```
>>> marimba.short_instrument_name_markup
Markup(('Mb.',))
```

```
>>> show(marimba.short_instrument_name_markup)
```

Mb.

Returns markup.

`Marimba.sounding_pitch_of_written_middle_c`

Gets sounding pitch of marimba's written middle C.

```
>>> marimba.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(marimba.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

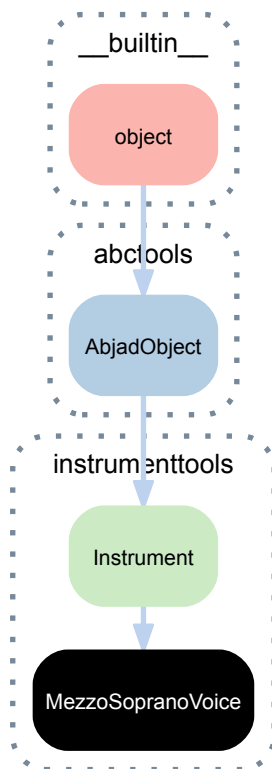
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

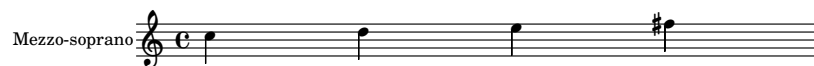
5.1.34 instrumenttools.MezzoSopranoVoice



```
class instrumenttools.MezzoSopranoVoice (instrument_name='mezzo-soprano',
                                         short_instrument_name='mezz.',      in-
                                         instrument_name_markup=None,
                                         short_instrument_name_markup=None,    al-
                                         allowable_clefs=None, pitch_range='[A3, C6]',
                                         sounding_pitch_of_written_middle_c=None)
```

A mezzo-soprano voice.

```
>>> staff = Staff("c''4 d''4 e''4 fs''4")
>>> mezzo_soprano = instrumenttools.MezzoSopranoVoice()
>>> attach(mezzo_soprano, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

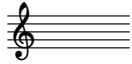
Read-only properties

MezzoSopranoVoice.allowable_clefs

Gets mezzo-soprano's allowable clefs.

```
>>> mezzo_soprano.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(mezzo_soprano.allowable_clefs)
```



Returns clef inventory.

MezzoSopranoVoice.instrument_name

Gets mezzo-soprano's name.

```
>>> mezzo_soprano.instrument_name
'mezzo-soprano'
```

Returns string.

MezzoSopranoVoice.instrument_name_markup

Gets mezzo-soprano's instrument name markup.

```
>>> mezzo_soprano.instrument_name_markup
Markup(('Mezzo-soprano',))
```

```
>>> show(mezzo_soprano.instrument_name_markup)
```

Mezzo-soprano

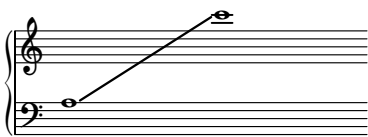
Returns markup.

MezzoSopranoVoice.pitch_range

Gets mezzo-soprano's range.

```
>>> mezzo_soprano.pitch_range
PitchRange(' [A3, C6]')
```

```
>>> show(mezzo_soprano.pitch_range)
```



Returns pitch range.

MezzoSopranoVoice.short_instrument_name

Gets mezzo-soprano's short instrument name.

```
>>> mezzo_soprano.short_instrument_name
'mezz.'
```

Returns string.

MezzoSopranoVoice.short_instrument_name_markup

Gets mezzo-soprano's short instrument name markup.

```
>>> mezzo_soprano.short_instrument_name_markup
Markup(('Mezz.',))
```

```
>>> show(mezzo_soprano.short_instrument_name_markup)
```

Mezz.

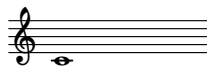
Returns markup.

`MezzoSopranoVoice.sounding_pitch_of_written_middle_c`

Gets sounding pitch of mezzo-soprano's written middle C.

```
>>> mezzo_soprano.sounding_pitch_of_written_middle_c
NamedPitch("c")
```

```
>>> show(mezzo_soprano.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

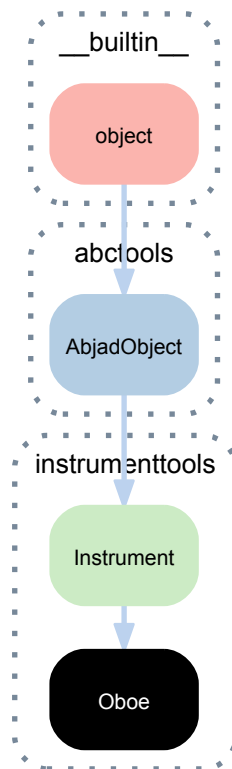
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

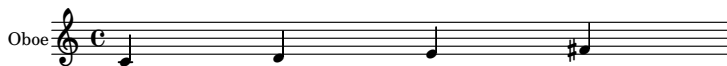
5.1.35 instrumenttools.Oboe



```
class instrumenttools.Oboe (instrument_name='oboe', short_instrument_name='ob.',
                           instrument_name_markup=None, short_instrument_name_markup=None,
                           allowable_clefs=None, pitch_range='[Bb3, A6]',
                           sounding_pitch_of_written_middle_c=None)
```

An oboe.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> oboe = instrumenttools.Oboe()
>>> attach(oboe, staff)
>>> show(staff)
```



Bases

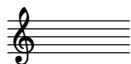
- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`Oboe.allowable_clefs`
Gets oboe's allowable clefs.

```
>>> oboe.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(oboe.allowable_clefs)
```



Returns clef inventory.

Oboe.**instrument_name**
Gets oboe's name.

```
>>> oboe.instrument_name
'oboe'
```

Returns string.

Oboe.**instrument_name_markup**
Gets oboe's instrument name markup.

```
>>> oboe.instrument_name_markup
Markup(('Oboe',))
```

```
>>> show(oboe.instrument_name_markup)
```

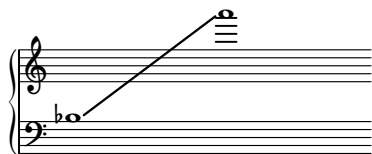
Oboe

Returns markup.

Oboe.**pitch_range**
Gets oboe's range.

```
>>> oboe.pitch_range
PitchRange(' [Bb3, A6]')
```

```
>>> show(oboe.pitch_range)
```



Returns pitch range.

Oboe.**short_instrument_name**
Gets oboe's short instrument name.

```
>>> oboe.short_instrument_name
'ob.'
```

Returns string.

Oboe.**short_instrument_name_markup**
Gets oboe's short instrument name markup.

```
>>> oboe.short_instrument_name_markup
Markup(('Ob.',))
```

```
>>> show(oboe.short_instrument_name_markup)
```

Ob.

Returns markup.

Oboe.**sounding_pitch_of_written_middle_c**
Gets sounding pitch of oboe's written middle C.

```
>>> oboe.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(oboe.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

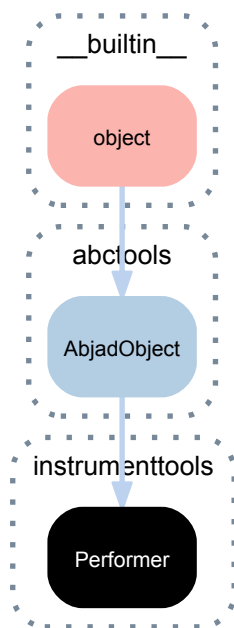
Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.36 instrumenttools.Performer



class `instrumenttools.Performer` (*name=None, instruments=None*)
 A performer.

```
>>> performer = instrumenttools.Performer(name='flutist')
>>> performer.instruments.append(instrumenttools.Flute())
>>> performer.instruments.append(instrumenttools.Piccolo())
```

```
>>> print format(performer)
instrumenttools.Performer(
  name='flutist',
  instruments=instrumenttools.InstrumentInventory(
    [
      instrumenttools.Flute(
        instrument_name='flute',
        short_instrument_name='fl.',
        instrument_name_markup=markuptools.Markup(
          ('Flute',)
        ),
        short_instrument_name_markup=markuptools.Markup(
          ('Fl.',)
        ),
        allowable_clefs=indicatortools.ClefInventory(
          [
            indicatortools.Clef(
              'treble'
            ),
          ]
        ),
        pitch_range=pitchtools.PitchRange(
          '[C4, D7]'
        ),
        sounding_pitch_of_written_middle_c=pitchtools.NamedPitch("c'"),
      ),
      instrumenttools.Piccolo(
        instrument_name='piccolo',
        short_instrument_name='picc.',
        instrument_name_markup=markuptools.Markup(
          ('Piccolo',)
        ),
        short_instrument_name_markup=markuptools.Markup(
          ('Picc.',)
        ),
        allowable_clefs=indicatortools.ClefInventory(
          [
```

```

        indicatortools.Clef(
            'treble'
        ),
    ],
    pitch_range=pitchtools.PitchRange(
        'D5, C8'
    ),
    sounding_pitch_of_written_middle_c=pitchtools.NamedPitch("c'"),
),
],
),
)

```

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`

True when performer is to play more than one instrument. Otherwise false.

```
::
```

```
>>> 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'>
```

Returns instrument class.

Read/write properties

`Performer.instruments`

Gets and sets instruments to be played by performer.

```
>>> for instrument in performer.instruments:
...     instrument
Flute()
Piccolo()
```

Returns instrument inventory.

`Performer.name`

Gets and sets score name of performer.

```
>>> performer.name
'flutist'
```

Returns string.

Methods

`Performer.make_performer_name_instrument_dictionary()`

Makes 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
alto:
    AltoVoice
baritone:
    BaritoneVoice
bass:
    BassVoice
bassist:
    Contrabass
bassoonist:
    Bassoon
    Contrabassoon
brass player:
    AltoTrombone
    BassTrombone
    FrenchHorn
    TenorTrombone
    Trumpet
    Tuba
cellist:
    Cello
clarinetist:
    BassClarinet
    ClarinetInA
    ClarinetInBFlat
    ClarinetInEFlat
    ContrabassClarinet
clarinettist:
    BassClarinet
    ClarinetInA
    ClarinetInBFlat
    ClarinetInEFlat
    ContrabassClarinet
contrabassist:
    Contrabass
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
    AltoVoice
    BaritoneSaxophone
    BaritoneVoice
    BassClarinet
    BassFlute
    Bassoon
    BassSaxophone
    BassTrombone
    BassVoice
    Cello
    ClarinetInA
    ClarinetInBFlat
    ClarinetInEFlat
    Contrabass
    ContrabassClarinet
    ContrabassFlute
    Contrabassoon
    ContrabassSaxophone
    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
oboiist:
    EnglishHorn
    Oboe
percussionist:
    Glockenspiel
    Marimba
    UntunedPercussion
    Vibraphone
    Xylophone
pianist:
    Piano
reed player:
    AltoSaxophone
    BaritoneSaxophone
    BassClarinet
    Bassoon
    BassSaxophone
    ClarinetInA
    ClarinetInBFlat
    ClarinetInEFlat
    ContrabassClarinet
    Contrabassoon
    ContrabassSaxophone
    EnglishHorn
    Oboe
    SopraninoSaxophone
    SopranoSaxophone
    TenorSaxophone
saxophonist:
    AltoSaxophone
    BaritoneSaxophone
    BassSaxophone
    ContrabassSaxophone
    SopraninoSaxophone
    SopranoSaxophone
    TenorSaxophone
single reed player:
    AltoSaxophone
    BaritoneSaxophone
    BassClarinet
    BassSaxophone
    ClarinetInA
    ClarinetInBFlat
    ClarinetInEFlat
    ContrabassClarinet
    ContrabassSaxophone
    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:
    AltoVoice
    BaritoneVoice
    BassVoice
    MezzoSopranoVoice
    SopranoVoice
    TenorVoice
wind player:
    AltoFlute
    AltoSaxophone
    BaritoneSaxophone
    BassClarinet
    BassFlute
    Bassoon
    BassSaxophone
    ClarinetInA
    ClarinetInBFlat
    ClarinetInEFlat
    ContrabassClarinet
    ContrabassFlute
    Contrabassoon
    ContrabassSaxophone
    EnglishHorn
    Flute
    FrenchHorn
    Oboe
    Piccolo
    SopraninoSaxophone
    SopranoSaxophone
    TenorSaxophone
xylophonist:
    Xylophone

```

Returns ordered dictionary.

Static methods

`Performer.list_performer_names()`

Lists performer names.

```

>>> for name in instrumenttools.Performer.list_performer_names():
...     name
...
'accordionist'
'alto'
'baritone'
'bass'
'bassist'
'bassoonist'
'cellist'
'clarinetist'
'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 primary performer names.

```
>>> for pair in instrumenttools.Performer.list_primary_performer_names():
...     pair
...
('accordionist', 'acc.')
('alto', 'alto')
('baritone', 'bar.')
('bass', 'bass')
('bassist', 'vb.')
('bassoonist', 'bsn.')
('cellist', 'vc.')
('clarinetist', 'cl.')
('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', 'ten. trb.')
('trumpeter', 'tp.')
('tubist', 'tb.')
('violinist', 'vn.')
('violist', 'va.')
```

Returns list.

Special methods

`Performer.__eq__(expr)`

True when *expr* is a performer with name and instruments equal to those of this performer. Otherwise false.

Returns boolean.

`Performer.__format__(format_specification='')`

Formats performer.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`Performer.__hash__()`

Hashes performer.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

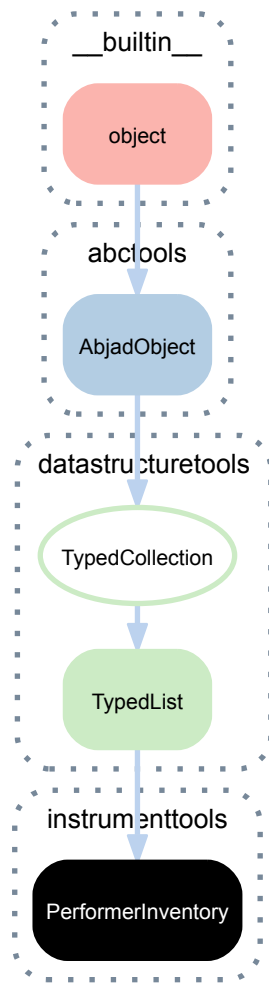
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

5.1.37 instrumenttools.PerformerInventory



class `instrumenttools.PerformerInventory` (*tokens=None*, *item_class=None*,
keep_sorted=None, *custom_identifier=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.

Read/write properties

(`TypedCollection`).**custom_identifier**
 Gets and sets custom identifier of typed collection.

Returns string or none.

(TypedList) **.keep_sorted**

Sorts collection on mutation if true.

Methods

(TypedList) **.append** (*token*)

Changes *token* to item and appends.

```
>>> 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*)

Changes *token* to item and returns 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*)

Changes *tokens* to items and extends.

```
>>> 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*)

Changes *token* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *token*)

Changes *token* to item and inserts.

```
>>> 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.

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

Aliases list.pop().

(TypedList) **.remove** (*token*)

Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) **.__format__** (*format_specification=''*)

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList) **.__getitem__** (*i*)

Aliases list.__getitem__().

(TypedList) **.__iadd__** (*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList (
  [
    indicatortools.Dynamic(
      'ppp'
    ),
    indicatortools.Dynamic(
      'p'
    ),
    indicatortools.Dynamic(
      'mp'
    ),
    indicatortools.Dynamic(
      'mf'
    ),
    indicatortools.Dynamic(
      'fff'
    ),
  ],
  item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList).**__reversed__**()

Aliases list.**__reversed__**().

(TypedList).**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

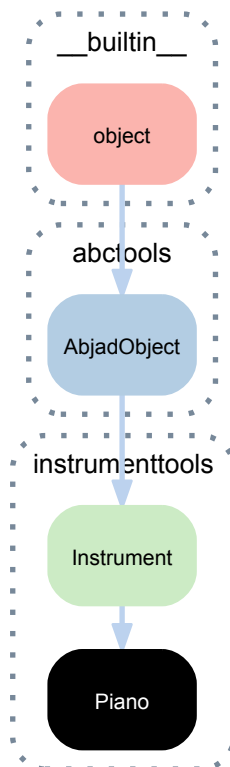
```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
datastructuretools.TypedList (
  [
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("d'"),
    pitchtools.NamedPitch("e'"),
    pitchtools.NamedPitch('gqs,')
  ],
  item_class=pitchtools.NamedPitch,
)
```

```
>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print format(pitch_collection)
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,
)
```

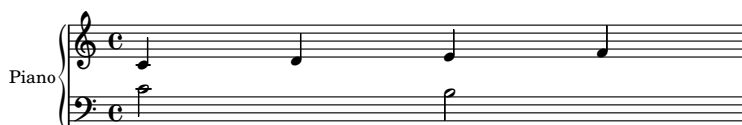
5.1.38 instrumenttools.Piano



```
class instrumenttools.Piano(instrument_name='piano', short_instrument_name='pf.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('treble', 'bass'), pitch_range='[A0, C8]', sounding_pitch_of_written_middle_c=None)
```

A piano.

```
>>> piano_staff = scoretools.PianoStaff()
>>> piano_staff.append(Staff("c'4 d'4 e'4 f'4"))
>>> piano_staff.append(Staff("c'2 b2"))
>>> piano = instrumenttools.Piano()
>>> attach(piano, piano_staff)
>>> attach(Clef('bass'), piano_staff[1])
>>> show(piano_staff)
```



The piano targets piano staff context by default.

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`Piano.allowable_clefs`

Gets piano's allowable clefs.

```
>>> piano.allowable_clefs
ClefInventory([Clef('treble'), Clef('bass')])
```

```
>>> show(piano.allowable_clefs)
```



Returns clef inventory.

`Piano.instrument_name`

Gets piano's name.

```
>>> piano.instrument_name
'piano'
```

Returns string.

`Piano.instrument_name_markup`

Gets piano's instrument name markup.

```
>>> piano.instrument_name_markup
Markup(('Piano',))
```

```
>>> show(piano.instrument_name_markup)
```

Piano

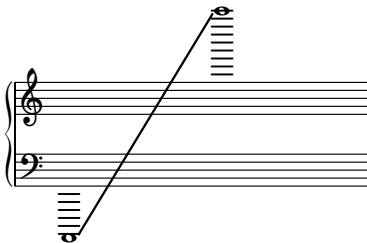
Returns markup.

`Piano.pitch_range`

Gets piano's range.

```
>>> piano.pitch_range
PitchRange(' [A0, C8]')
```

```
>>> show(piano.pitch_range)
```



Returns pitch range.

`Piano.short_instrument_name`

Gets piano's short instrument name.

```
>>> piano.short_instrument_name
'pf.'
```

Returns string.

Piano.short_instrument_name_markup
Gets piano's short instrument name markup.

```
>>> piano.short_instrument_name_markup
Markup(('Pf.',))
```

```
>>> show(piano.short_instrument_name_markup)
```

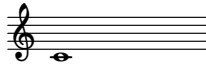
Pf.

Returns markup.

Piano.sounding_pitch_of_written_middle_c
Gets sounding pitch of piano's written middle C.

```
>>> piano.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(piano.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument).__copy__(*args)
Copies instrument.

Returns new instrument.

(Instrument).__eq__(arg)
Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument).__format__(format_specification='')
Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument).__hash__()
Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject).__ne__(expr)
Is true when Abjad object does not equal *expr*. Otherwise false.

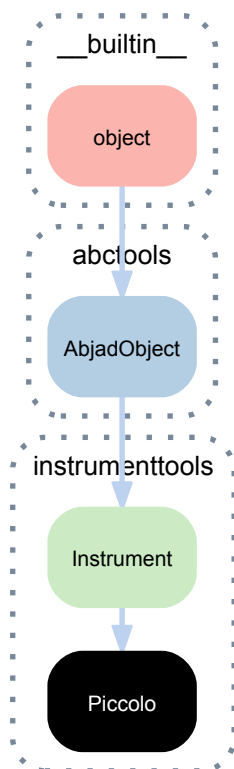
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

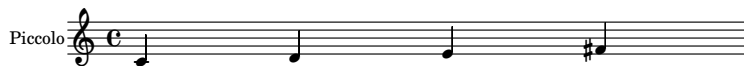
5.1.39 instrumenttools.Piccolo



```
class instrumenttools.Piccolo(instrument_name='piccolo', short_instrument_name='picc.',
                             instrument_name_markup=None,
                             short_instrument_name_markup=None, allow_
                             able_clefs=None, pitch_range='[D5, C8]', sound-
                             ing_pitch_of_written_middle_c='C5')
```

A piccolo.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> piccolo = instrumenttools.Piccolo()
>>> attach(piccolo, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

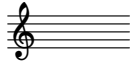
Read-only properties

Piccolo.allowable_clefs

Gets piccolo's allowable clefs.

```
>>> piccolo.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(piccolo.allowable_clefs)
```



Returns clef inventory.

Piccolo.instrument_name

Gets piccolo's name.

```
>>> piccolo.instrument_name
'piccolo'
```

Returns string.

Piccolo.instrument_name_markup

Gets piccolo's instrument name markup.

```
>>> piccolo.instrument_name_markup
Markup(('Piccolo',))
```

```
>>> show(piccolo.instrument_name_markup)
```

Piccolo

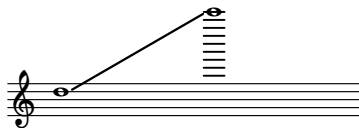
Returns markup.

Piccolo.pitch_range

Gets piccolo's range.

```
>>> piccolo.pitch_range
PitchRange(' [D5, C8]')
```

```
>>> show(piccolo.pitch_range)
```



Returns pitch range.

Piccolo.short_instrument_name

Gets piccolo's short instrument name.

```
>>> piccolo.short_instrument_name
'picc.'
```

Returns string.

Piccolo.short_instrument_name_markup

Gets piccolo's short instrument name markup.

```
>>> piccolo.short_instrument_name_markup
Markup(('Picc.',))
```

```
>>> show(piccolo.short_instrument_name_markup)
```

Picc.

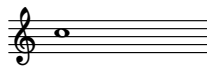
Returns markup.

`Piccolo.sounding_pitch_of_written_middle_c`

Gets sounding pitch of piccolo's written middle C.

```
>>> piccolo.sounding_pitch_of_written_middle_c
NamedPitch('c''')
```

```
>>> show(piccolo.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

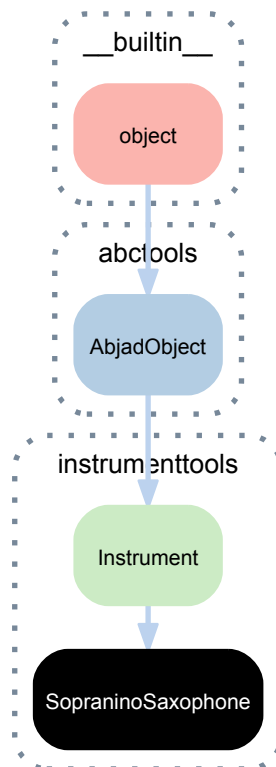
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.40 instrumenttools.SopraninoSaxophone



```

class instrumenttools.SopraninoSaxophone (instrument_name='sopranino saxophone',
                                           short_instrument_name='sopranino sax.',
                                           instrument_name_markup=None,
                                           short_instrument_name_markup=None,
                                           allowable_clefs=None,
                                           pitch_range='[Db4, F#6]',
                                           sounding_pitch_of_written_middle_c='Eb4')
  
```

A sopranino saxophone.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> sopranino_saxophone = instrumenttools.SopraninoSaxophone()
>>> attach(sopranino_saxophone, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

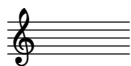
Read-only properties

`SopraninoSaxophone.allowable_clefs`
Gets sopranino saxophone's allowable clefs.

```

>>> sopranino_saxophone.allowable_clefs
ClefInventory([Clef('treble')])
  
```

```
>>> show(sopranino_saxophone.allowable_clefs)
```



Returns clef inventory.

SopraninoSaxophone.**instrument_name**

Gets soprano saxophone's name.

```
>>> sopranino_saxophone.instrument_name
'sopranino saxophone'
```

Returns string.

SopraninoSaxophone.**instrument_name_markup**

Gets soprano saxophone's instrument name markup.

```
>>> sopranino_saxophone.instrument_name_markup
Markup(('Sopranino saxophone',))
```

```
>>> show(sopranino_saxophone.instrument_name_markup)
```

Sopranino saxophone

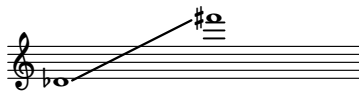
Returns markup.

SopraninoSaxophone.**pitch_range**

Gets soprano saxophone's range.

```
>>> sopranino_saxophone.pitch_range
PitchRange(' [Db4, F#6]')
```

```
>>> show(sopranino_saxophone.pitch_range)
```



Returns pitch range.

SopraninoSaxophone.**short_instrument_name**

Gets soprano saxophone's short instrument name.

```
>>> sopranino_saxophone.short_instrument_name
'sopranino sax.'
```

Returns string.

SopraninoSaxophone.**short_instrument_name_markup**

Gets soprano saxophone's short instrument name markup.

```
>>> sopranino_saxophone.short_instrument_name_markup
Markup(('Sopranino sax.',))
```

```
>>> show(sopranino_saxophone.short_instrument_name_markup)
```

Sopranino sax.

Returns markup.

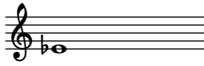
SopraninoSaxophone.**sounding_pitch_of_written_middle_c**

Gets sounding pitch of soprano saxophone's written middle C.

```
>>> sopranino_saxophone.sounding_pitch_of_written_middle_c
NamedPitch('ef')
```



```
>>> show(sopranino_saxophone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__(*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__(format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

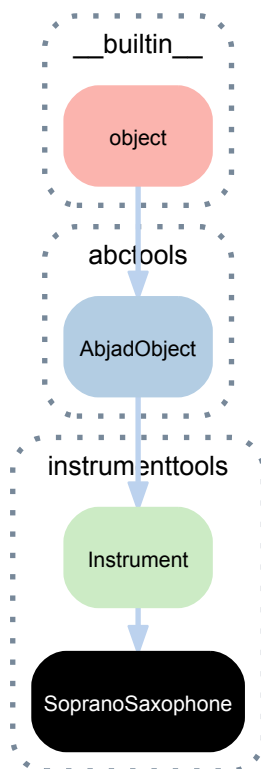
Returns boolean.

(Instrument) .__repr__()

Gets interpreter representation of instrument.

Returns string.

5.1.41 instrumenttools.SopranoSaxophone



class `instrumenttools.SopranoSaxophone` (*instrument_name*=*'soprano saxophone'*, *short_instrument_name*=*'sop. sax.'*, *instrument_name_markup*=*None*, *short_instrument_name_markup*=*None*, *allowable_clefs*=*None*, *pitch_range*=*'[Ab3, E6]'*, *sounding_pitch_of_written_middle_c*=*'Bb3'*)

A soprano saxophone.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> soprano_saxophone = instrumenttools.SopranoSaxophone()
>>> attach(soprano_saxophone, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

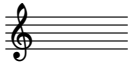
Read-only properties

`SopranoSaxophone.allowable_clefs`
Gets soprano saxophone's allowable clefs.

```

>>> soprano_saxophone.allowable_clefs
ClefInventory([Clef('treble')])
  
```

```
>>> show(soprano_saxophone.allowable_clefs)
```



Returns clef inventory.

SopranoSaxophone.instrument_name

Gets soprano saxophone's name.

```
>>> soprano_saxophone.instrument_name
'soprano saxophone'
```

Returns string.

SopranoSaxophone.instrument_name_markup

Gets soprano saxophone's instrument name markup.

```
>>> soprano_saxophone.instrument_name_markup
Markup(('Soprano saxophone',))
```

```
>>> show(soprano_saxophone.instrument_name_markup)
```

Soprano saxophone

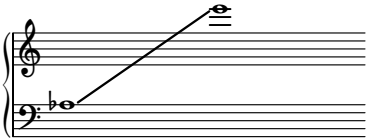
Returns markup.

SopranoSaxophone.pitch_range

Gets soprano saxophone's range.

```
>>> soprano_saxophone.pitch_range
PitchRange(' [Ab3, E6]')
```

```
>>> show(soprano_saxophone.pitch_range)
```



Returns pitch range.

SopranoSaxophone.short_instrument_name

Gets soprano saxophone's short instrument name.

```
>>> soprano_saxophone.short_instrument_name
'sop. sax.'
```

Returns string.

SopranoSaxophone.short_instrument_name_markup

Gets soprano saxophone's short instrument name markup.

```
>>> soprano_saxophone.short_instrument_name_markup
Markup(('Sop. sax.',))
```

```
>>> show(soprano_saxophone.short_instrument_name_markup)
```

Sop. sax.

Returns markup.

SopranoSaxophone.sounding_pitch_of_written_middle_c

Gets sounding pitch of soprano saxophone's written middle C.

```
>>> soprano_saxophone.sounding_pitch_of_written_middle_c
NamedPitch('bf')
```

```
>>> show(soprano_saxophone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__** (*args)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__** (arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__** (format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__** ()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__** (instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__** (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

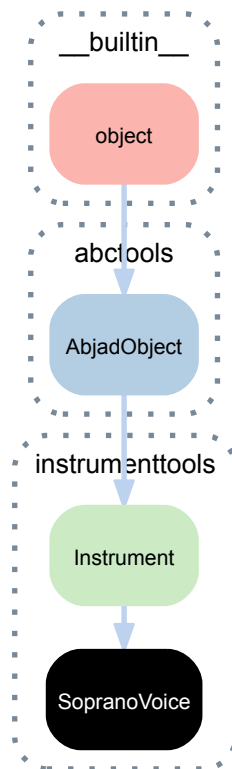
Returns boolean.

(Instrument) .**__repr__** ()

Gets interpreter representation of instrument.

Returns string.

5.1.42 instrumenttools.SopranoVoice



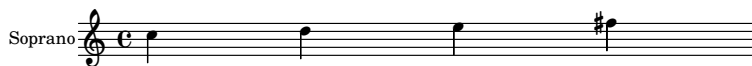
```

class instrumenttools.SopranoVoice (instrument_name='soprano',
                                     short_instrument_name='sop.',          in-
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,      allow-
                                     allowable_clefs=None, pitch_range='[C4, E6]', sound-
                                     ing_pitch_of_written_middle_c=None)
  
```

A soprano voice.

```

>>> staff = Staff("c''4 d''4 e''4 fs''4")
>>> soprano = instrumenttools.SopranoVoice()
>>> attach(soprano, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

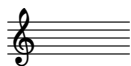
Read-only properties

`SopranoVoice.allowable_clefs`
Gets soprano's allowable clefs.

```

>>> soprano.allowable_clefs
ClefInventory([Clef('treble')])
  
```

```
>>> show(soprano.allowable_clefs)
```



Returns clef inventory.

SopranoVoice.instrument_name

Gets soprano's name.

```
>>> soprano.instrument_name
'soprano'
```

Returns string.

SopranoVoice.instrument_name_markup

Gets soprano's instrument name markup.

```
>>> soprano.instrument_name_markup
Markup(('Soprano',))
```

```
>>> show(soprano.instrument_name_markup)
```

Soprano

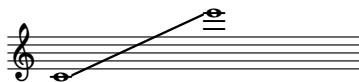
Returns markup.

SopranoVoice.pitch_range

Gets soprano's range.

```
>>> soprano.pitch_range
PitchRange('C4, E6')
```

```
>>> show(soprano.pitch_range)
```



Returns pitch range.

SopranoVoice.short_instrument_name

Gets soprano's short instrument name.

```
>>> soprano.short_instrument_name
'sop.'
```

Returns string.

SopranoVoice.short_instrument_name_markup

Gets soprano's short instrument name markup.

```
>>> soprano.short_instrument_name_markup
Markup(('Sop.',))
```

```
>>> show(soprano.short_instrument_name_markup)
```

Sop.

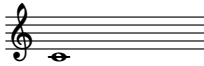
Returns markup.

SopranoVoice.sounding_pitch_of_written_middle_c

Gets sounding pitch of soprano's written middle C.

```
>>> soprano.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(soprano.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__(*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__(format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

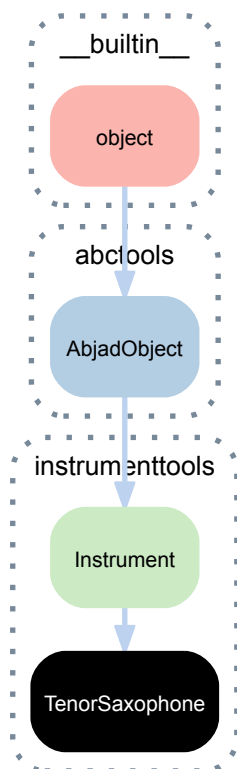
Returns boolean.

(Instrument) .__repr__()

Gets interpreter representation of instrument.

Returns string.

5.1.43 instrumenttools.TenorSaxophone



```

class instrumenttools.TenorSaxophone (instrument_name='tenor saxophone',
                                       short_instrument_name='ten. sax.',
                                       instrument_name_markup=None,
                                       short_instrument_name_markup=None,
                                       allowable_clefs=None, pitch_range='[Ab2, E5]', sound-
                                       ing_pitch_of_written_middle_c='Bb2')

```

A tenor saxophone.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> tenor_saxophone = instrumenttools.TenorSaxophone()
>>> attach(tenor_saxophone, staff)
>>> show(staff)

```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`TenorSaxophone.allowable_clefs`
Gets tenor saxophone's allowable clefs.

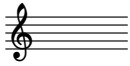
```

>>> tenor_saxophone.allowable_clefs
ClefInventory([Clef('treble')])

```



```
>>> show(tenor_saxophone.allowable_clefs)
```



Returns clef inventory.

TenorSaxophone.instrument_name

Gets tenor saxophone's name.

```
>>> tenor_saxophone.instrument_name
'tenor saxophone'
```

Returns string.

TenorSaxophone.instrument_name_markup

Gets tenor saxophone's instrument name markup.

```
>>> tenor_saxophone.instrument_name_markup
Markup(('Tenor saxophone',))
```

```
>>> show(tenor_saxophone.instrument_name_markup)
```

Tenor saxophone

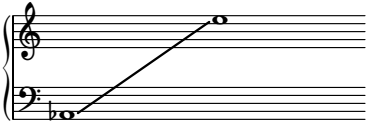
Returns markup.

TenorSaxophone.pitch_range

Gets tenor saxophone's range.

```
>>> tenor_saxophone.pitch_range
PitchRange(' [Ab2, E5]')
```

```
>>> show(tenor_saxophone.pitch_range)
```



Returns pitch range.

TenorSaxophone.short_instrument_name

Gets tenor saxophone's short instrument name.

```
>>> tenor_saxophone.short_instrument_name
'ten. sax.'
```

Returns string.

TenorSaxophone.short_instrument_name_markup

Gets tenor saxophone's short instrument name markup.

```
>>> tenor_saxophone.short_instrument_name_markup
Markup(('Ten. sax.',))
```

```
>>> show(tenor_saxophone.short_instrument_name_markup)
```

Ten. sax.

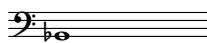
Returns markup.

TenorSaxophone.sounding_pitch_of_written_middle_c

Gets sounding pitch of tenor saxophone's written middle C.

```
>>> tenor_saxophone.sounding_pitch_of_written_middle_c
NamedPitch('bf,')
```

```
>>> show(tenor_saxophone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

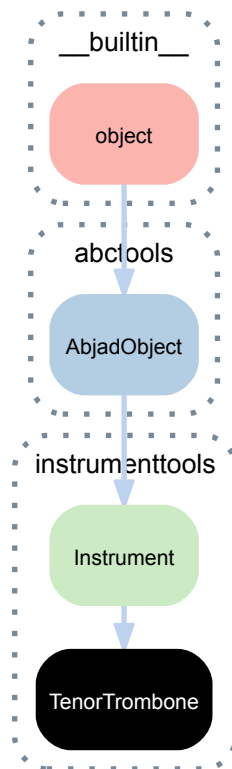
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.44 instrumenttools.TenorTrombone



```
class instrumenttools.TenorTrombone (instrument_name='tenor          trombone',
                                     short_instrument_name='ten.          trb.',
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,          allow-
                                     able_clefs=('tenor', 'bass'), pitch_range='[E2, Eb5]',
                                     sounding_pitch_of_written_middle_c=None)
```

A tenor trombone.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> tenor_trombone = instrumenttools.TenorTrombone()
>>> attach(tenor_trombone, staff)
>>> show(staff)
```



Bases

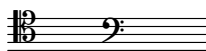
- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`TenorTrombone.allowable_clefs`
Gets tenor trombone's allowable clefs.

```
>>> tenor_trombone.allowable_clefs
ClefInventory([Clef('tenor'), Clef('bass')])
```

```
>>> show(tenor_trombone.allowable_clefs)
```



Returns clef inventory.

TenorTrombone.instrument_name
Gets tenor trombone's name.

```
>>> tenor_trombone.instrument_name
'tenor trombone'
```

Returns string.

TenorTrombone.instrument_name_markup
Gets tenor trombone's instrument name markup.

```
>>> tenor_trombone.instrument_name_markup
Markup(('Tenor trombone',))
```

```
>>> show(tenor_trombone.instrument_name_markup)
```

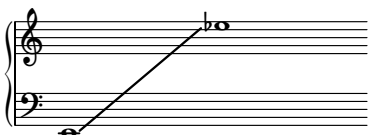
Tenor trombone

Returns markup.

TenorTrombone.pitch_range
Gets tenor trombone's range.

```
>>> tenor_trombone.pitch_range
PitchRange(' [E2, Eb5]')
```

```
>>> show(tenor_trombone.pitch_range)
```



Returns pitch range.

TenorTrombone.short_instrument_name
Gets tenor trombone's short instrument name.

```
>>> tenor_trombone.short_instrument_name
'ten. trb.'
```

Returns string.

TenorTrombone.short_instrument_name_markup
Gets tenor trombone's short instrument name markup.

```
>>> tenor_trombone.short_instrument_name_markup
Markup(('Ten. trb.',))
```

```
>>> show(tenor_trombone.short_instrument_name_markup)
```

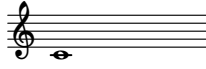
Ten. trb.

Returns markup.

TenorTrombone.sounding_pitch_of_written_middle_c
Gets sounding pitch of tenor trombone's written middle C.

```
>>> tenor_trombone.sounding_pitch_of_written_middle_c
NamedPitch("c")
```

```
>>> show(tenor_trombone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument).__copy__(*args)

Copies instrument.

Returns new instrument.

(Instrument).__eq__(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument).__format__(format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument).__hash__()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject).__ne__(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

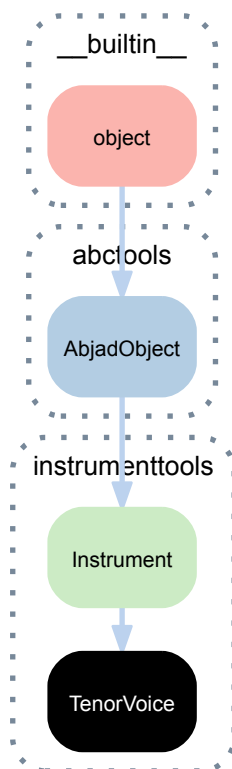
Returns boolean.

(Instrument).__repr__()

Gets interpreter representation of instrument.

Returns string.

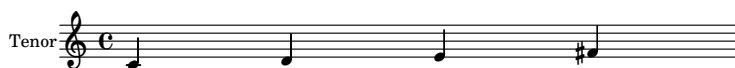
5.1.45 instrumenttools.TenorVoice



```
class instrumenttools.TenorVoice (instrument_name='tenor', short_instrument_name='ten.',
                                   instrument_name_markup=None,
                                   short_instrument_name_markup=None, allow-
                                   able_clefs=None, pitch_range='[C3, D5]', sound-
                                   ing_pitch_of_written_middle_c=None)
```

A tenor voice.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> tenor = instrumenttools.TenorVoice()
>>> attach(tenor, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

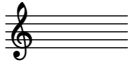
Read-only properties

`TenorVoice.allowable_clefs`

Gets tenor's allowable clefs.

```
>>> tenor.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(tenor.allowable_clefs)
```



Returns clef inventory.

TenorVoice.instrument_name
Gets tenor's name.

```
>>> tenor.instrument_name
'tenor'
```

Returns string.

TenorVoice.instrument_name_markup
Gets tenor's instrument name markup.

```
>>> tenor.instrument_name_markup
Markup(('Tenor',))
```

```
>>> show(tenor.instrument_name_markup)
```

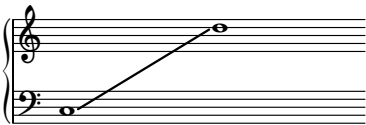
Tenor

Returns markup.

TenorVoice.pitch_range
Gets tenor's range.

```
>>> tenor.pitch_range
PitchRange(' [C3, D5]')
```

```
>>> show(tenor.pitch_range)
```



Returns pitch range.

TenorVoice.short_instrument_name
Gets tenor's short instrument name.

```
>>> tenor.short_instrument_name
'ten.'
```

Returns string.

TenorVoice.short_instrument_name_markup
Gets tenor's short instrument name markup.

```
>>> tenor.short_instrument_name_markup
Markup(('Ten.',))
```

```
>>> show(tenor.short_instrument_name_markup)
```

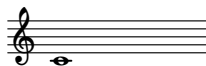
Ten.

Returns markup.

TenorVoice.sounding_pitch_of_written_middle_c
Gets sounding pitch of tenor's written middle C.

```
>>> tenor.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(tenor.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

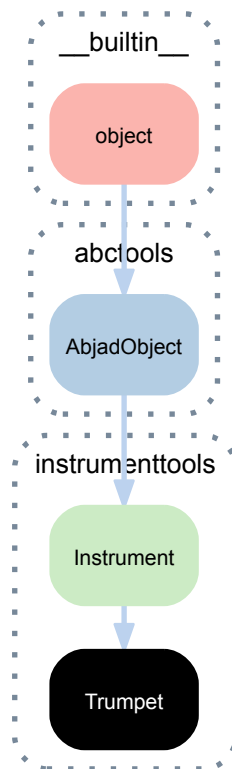
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

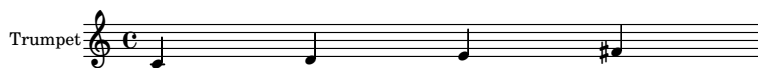
5.1.46 instrumenttools.Trumpet



```
class instrumenttools.Trumpet (instrument_name='trumpet',      short_instrument_name='tp.',
                              instrument_name_markup=None,
                              short_instrument_name_markup=None,      allow-
                              able_clefs=None,      pitch_range='[F#3,      D6]',      sound-
                              ing_pitch_of_written_middle_c=None)
```

A trumpet.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> trumpet = instrumenttools.Trumpet()
>>> attach(trumpet, staff)
>>> show(staff)
```



Bases

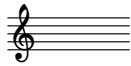
- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `___builtin___object`

Read-only properties

`Trumpet.allowable_clefs`
Gets trumpet's allowable clefs.

```
>>> trumpet.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(trumpet.allowable_clefs)
```



Returns clef inventory.

Trumpet.instrument_name
Gets trumpet's name.

```
>>> trumpet.instrument_name  
'trumpet'
```

Returns string.

Trumpet.instrument_name_markup
Gets trumpet's instrument name markup.

```
>>> trumpet.instrument_name_markup  
Markup(('Trumpet',))
```

```
>>> show(trumpet.instrument_name_markup)
```

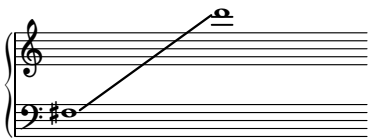
Trumpet

Returns markup.

Trumpet.pitch_range
Gets trumpet's range.

```
>>> trumpet.pitch_range  
PitchRange(' [F#3, D6]')
```

```
>>> show(trumpet.pitch_range)
```



Returns pitch range.

Trumpet.short_instrument_name
Gets trumpet's short instrument name.

```
>>> trumpet.short_instrument_name  
'tp.'
```

Returns string.

Trumpet.short_instrument_name_markup
Gets trumpet's short instrument name markup.

```
>>> trumpet.short_instrument_name_markup  
Markup(('Tp.',))
```

```
>>> show(trumpet.short_instrument_name_markup)
```

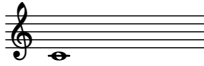
Tp.

Returns markup.

Trumpet.sounding_pitch_of_written_middle_c
Gets sounding pitch of trumpet's written middle C.

```
>>> trumpet.sounding_pitch_of_written_middle_c  
NamedPitch('c')
```

```
>>> show(trumpet.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

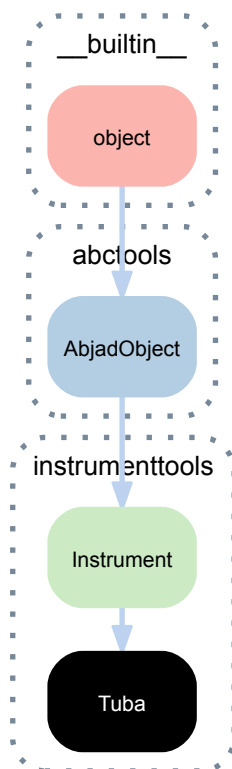
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

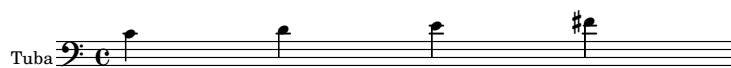
5.1.47 instrumenttools.Tuba



class instrumenttools.Tuba(*instrument_name='tuba', short_instrument_name='tb.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('bass',), pitch_range='[D1, F4]', sounding_pitch_of_written_middle_c=None*)

A tuba.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef('bass')
>>> attach(clef, staff)
>>> tuba = instrumenttools.Tuba()
>>> attach(tuba, staff)
>>> show(staff)
```



Bases

- instrumenttools.Instrument
- abctools.AbjadObject
- ___builtin___object

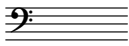
Read-only properties

Tuba.**allowable_clefs**

Gets tuba's allowable clefs.

```
>>> tuba.allowable_clefs
ClefInventory([Clef('bass')])
```

```
>>> show(tuba.allowable_clefs)
```



Returns clef inventory.

Tuba.**instrument_name**
Gets tuba's name.

```
>>> tuba.instrument_name
'tuba'
```

Returns string.

Tuba.**instrument_name_markup**
Gets tuba's instrument name markup.

```
>>> tuba.instrument_name_markup
Markup(('Tuba',))
```

```
>>> show(tuba.instrument_name_markup)
```

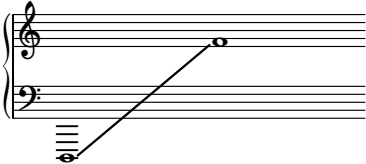
Tuba

Returns markup.

Tuba.**pitch_range**
Gets tuba's range.

```
>>> tuba.pitch_range
PitchRange('[D1, F4]')
```

```
>>> show(tuba.pitch_range)
```



Returns pitch range.

Tuba.**short_instrument_name**
Gets tuba's short instrument name.

```
>>> tuba.short_instrument_name
'tb.'
```

Returns string.

Tuba.**short_instrument_name_markup**
Gets tuba's short instrument name markup.

```
>>> tuba.short_instrument_name_markup
Markup(('Tb.',))
```

```
>>> show(tuba.short_instrument_name_markup)
```

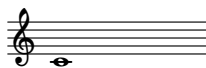
Tb.

Returns markup.

Tuba.**sounding_pitch_of_written_middle_c**
Gets sounding pitch of tuba's written middle C.

```
>>> tuba.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(tuba.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__(*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__(format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

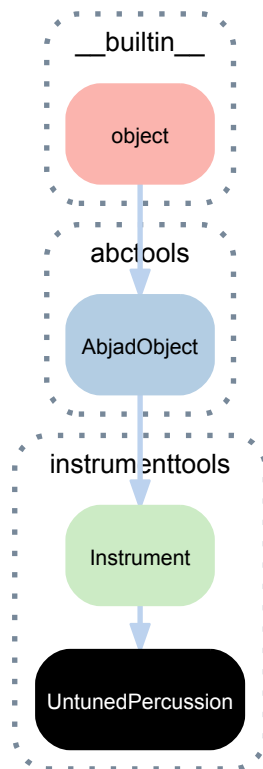
Returns boolean.

(Instrument) .__repr__()

Gets interpreter representation of instrument.

Returns string.

5.1.48 instrumenttools.UntunedPercussion



class instrumenttools.UntunedPercussion (*instrument_name='untuned percussion', short_instrument_name='perc.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('percussion',), pitch_range=None, sounding_pitch_of_written_middle_c=None*)

An untuned percussion instrument.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> untuned_percussion = instrumenttools.UntunedPercussion()
>>> attach(untuned_percussion, staff)
>>> show(staff)
  
```

Untuned percussion

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

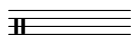
Read-only properties

UntunedPercussion.allowable_clefs
Gets untuned percussion's allowable clefs.

```

>>> untuned_percussion.allowable_clefs
ClefInventory([Clef('percussion')])
  
```

```
>>> show(untuned_percussion.allowable_clefs)
```



Returns clef inventory.

UntunedPercussion.instrument_name

Gets untuned percussion's name.

```
>>> untuned_percussion.instrument_name
'untuned percussion'
```

Returns string.

UntunedPercussion.instrument_name_markup

Gets untuned percussion's instrument name markup.

```
>>> untuned_percussion.instrument_name_markup
Markup(('Untuned percussion',))
```

```
>>> show(untuned_percussion.instrument_name_markup)
```

Untuned percussion

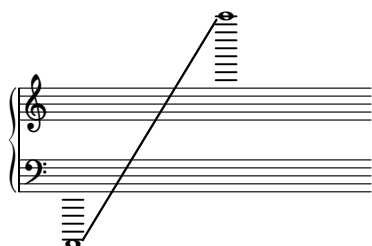
Returns markup.

UntunedPercussion.pitch_range

Gets untuned percussion's range.

```
>>> untuned_percussion.pitch_range
PitchRange(' [A0, C8]')
```

```
>>> show(untuned_percussion.pitch_range)
```



Returns pitch range.

UntunedPercussion.short_instrument_name

Gets untuned percussion's short instrument name.

```
>>> untuned_percussion.short_instrument_name
'perc.'
```

Returns string.

UntunedPercussion.short_instrument_name_markup

Gets untuned percussion's short instrument name markup.

```
>>> untuned_percussion.short_instrument_name_markup
Markup(('Perc.',))
```

```
>>> show(untuned_percussion.short_instrument_name_markup)
```

Perc.

Returns markup.

UntunedPercussion.sounding_pitch_of_written_middle_c

Gets sounding pitch of untuned percussion's written middle C.


```
>>> untuned_percussion.sounding_pitch_of_written_middle_c
NamedPitch("c' ")
```

```
>>> show(untuned_percussion.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

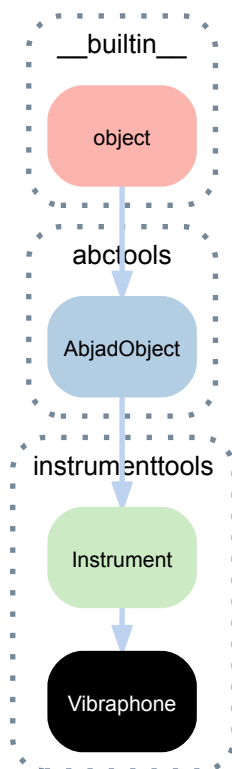
Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

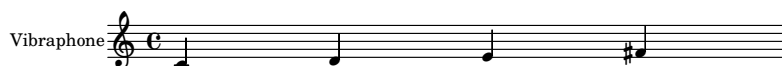
5.1.49 instrumenttools.Vibraphone



class instrumenttools.Vibraphone (*instrument_name='vibraphone',*
short_instrument_name='vibr', *in-*
strument_name_markup=None,
short_instrument_name_markup=None, *allow-*
able_clefs=None, *pitch_range='[F3, F6]',* *sound-*
ing_pitch_of_written_middle_c=None)

A vibraphone.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> vibraphone = instrumenttools.Vibraphone()
>>> attach(vibraphone, staff)
>>> show(staff)
```



Bases

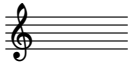
- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

Vibraphone.allowable_clefs
 Gets vibraphone's allowable clefs.

```
>>> vibraphone.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(vibraphone.allowable_clefs)
```



Returns clef inventory.

Vibraphone.instrument_name

Gets vibraphone's name.

```
>>> vibraphone.instrument_name
'vibraphone'
```

Returns string.

Vibraphone.instrument_name_markup

Gets vibraphone's instrument name markup.

```
>>> vibraphone.instrument_name_markup
Markup(('Vibraphone',))
```

```
>>> show(vibraphone.instrument_name_markup)
```

Vibraphone

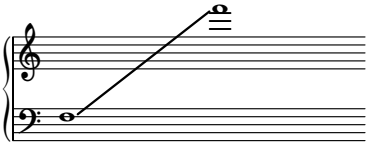
Returns markup.

Vibraphone.pitch_range

Gets vibraphone's range.

```
>>> vibraphone.pitch_range
PitchRange(' [F3, F6]')
```

```
>>> show(vibraphone.pitch_range)
```



Returns pitch range.

Vibraphone.short_instrument_name

Gets vibraphone's short instrument name.

```
>>> vibraphone.short_instrument_name
'vibr.'
```

Returns string.

Vibraphone.short_instrument_name_markup

Gets vibraphone's short instrument name markup.

```
>>> vibraphone.short_instrument_name_markup
Markup(('Vibr.',))
```

```
>>> show(vibraphone.short_instrument_name_markup)
```

Vibr.

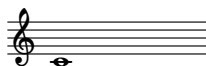
Returns markup.

Vibraphone.sounding_pitch_of_written_middle_c

Gets sounding pitch of vibraphone's written middle C.

```
>>> vibraphone.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(vibraphone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

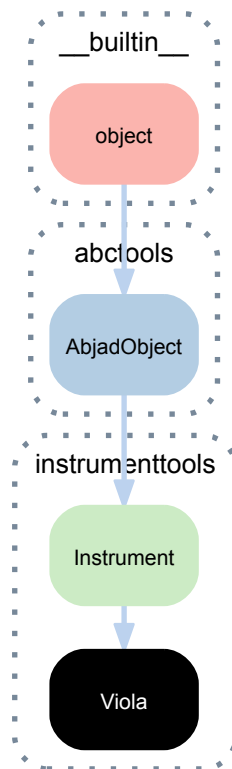
Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.50 instrumenttools.Viola

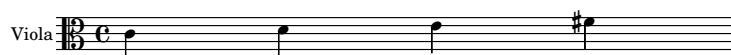


class instrumenttools.**Viola**(*instrument_name='viola', short_instrument_name='va.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('alto', 'treble'), pitch_range='[C3, D6]', sounding_pitch_of_written_middle_c=None*)

A viola.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef('alto')
>>> attach(clef, staff)
>>> viola = instrumenttools.Viola()
>>> attach(viola, staff)
>>> show(staff)
  
```



Bases

- instrumenttools.Instrument
- abctools.AbjadObject
- __builtin__.object

Read-only properties

Viola.allowable_clefs

Gets viola's allowable clefs.

```

>>> viola.allowable_clefs
ClefInventory([Clef('alto'), Clef('treble')])
  
```

```
>>> show(viola.allowable_clefs)
```



Returns clef inventory.

Viola.instrument_name

Gets viola's name.

```
>>> viola.instrument_name
'viola'
```

Returns string.

Viola.instrument_name_markup

Gets viola's instrument name markup.

```
>>> viola.instrument_name_markup
Markup(('Viola',))
```

```
>>> show(viola.instrument_name_markup)
```

Viola

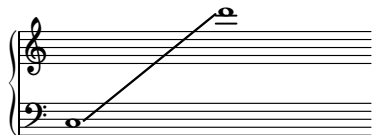
Returns markup.

Viola.pitch_range

Gets viola's range.

```
>>> viola.pitch_range
PitchRange(' [C3, D6]')
```

```
>>> show(viola.pitch_range)
```



Returns pitch range.

Viola.short_instrument_name

Gets viola's short instrument name.

```
>>> viola.short_instrument_name
'va.'
```

Returns string.

Viola.short_instrument_name_markup

Gets viola's short instrument name markup.

```
>>> viola.short_instrument_name_markup
Markup(('Va.',))
```

```
>>> show(viola.short_instrument_name_markup)
```

Va.

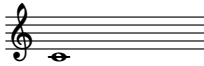
Returns markup.

Viola.sounding_pitch_of_written_middle_c

Gets sounding pitch of viola's written middle C.

```
>>> viola.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(viola.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

`(Instrument).__copy__(*args)`

Copies instrument.

Returns new instrument.

`(Instrument).__eq__(arg)`

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

`(Instrument).__format__(format_specification='')`

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

`(Instrument).__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)`

Makes new instrument.

Returns new instrument.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

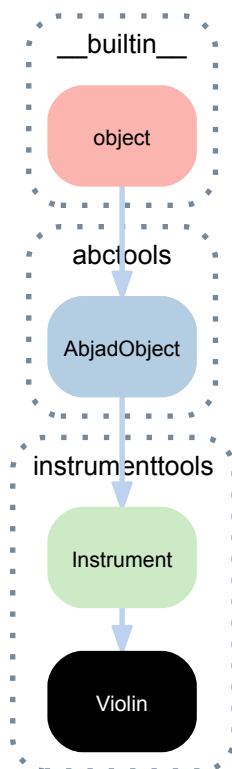
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

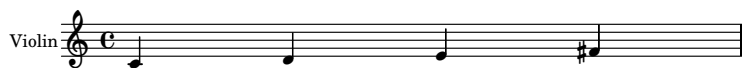
5.1.51 instrumenttools.Violin



class instrumenttools.**Violin**(*instrument_name='violin', short_instrument_name='vn.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[G3, G7]', sounding_pitch_of_written_middle_c=None*)

A violin.

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> violin = instrumenttools.Violin()
>>> attach(violin, staff)
>>> show(staff)
```



Bases

- instrumenttools.Instrument
- abctools.AbjadObject
- __builtin__.object

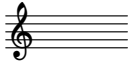
Read-only properties

Violin.allowable_clefs

Gets violin's allowable clefs.

```
>>> violin.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(violin.allowable_clefs)
```

Returns clef inventory.

Violin.instrument_name
Gets violin's name.

```
>>> violin.instrument_name
'violin'
```

Returns string.

Violin.instrument_name_markup
Gets violin's instrument name markup.

```
>>> violin.instrument_name_markup
Markup(('Violin',))
```

```
>>> show(violin.instrument_name_markup)
```

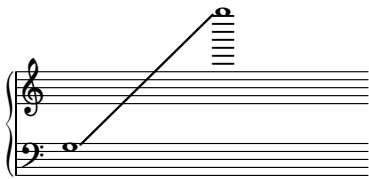
Violin

Returns markup.

Violin.pitch_range
Gets violin's range.

```
>>> violin.pitch_range
PitchRange(' [G3, G7]')
```

```
>>> show(violin.pitch_range)
```



Returns pitch range.

Violin.short_instrument_name
Gets violin's short instrument name.

```
>>> violin.short_instrument_name
'vn.'
```

Returns string.

Violin.short_instrument_name_markup
Gets violin's short instrument name markup.

```
>>> violin.short_instrument_name_markup
Markup(('Vn.',))
```

```
>>> show(violin.short_instrument_name_markup)
```

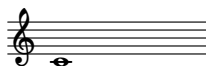
Vn.

Returns markup.

Violin.sounding_pitch_of_written_middle_c
Gets sounding pitch of violin's written middle C.

```
>>> violin.sounding_pitch_of_written_middle_c
NamedPitch('c')
```

```
>>> show(violin.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .__copy__(*args)

Copies instrument.

Returns new instrument.

(Instrument) .__eq__(arg)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .__format__(format_specification='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .__hash__()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .__makenew__(instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None)

Makes new instrument.

Returns new instrument.

(AbjadObject) .__ne__(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

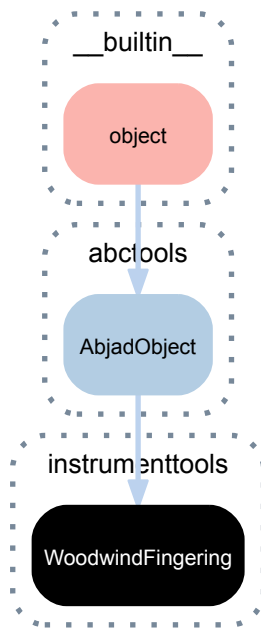
Returns boolean.

(Instrument) .__repr__()

Gets interpreter representation of instrument.

Returns string.

5.1.52 instrumenttools.WoodwindFingering



class `instrumenttools.WoodwindFingering` (*instrument_name=None*, *center_column=None*, *left_hand=None*, *right_hand=None*)

A woodwind fingering.

Initializes 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(
...     instrument_name='clarinet',
...     center_column=center_column,
...     left_hand=left_hand,
...     right_hand=right_hand,
... )
```

```
>>> print format(woodwind_fingering, 'storage')
instrumenttools.WoodwindFingering(
    instrument_name='clarinet',
    center_column=('one', 'two', 'three', 'five'),
    left_hand=('R', 'thumb'),
    right_hand=('e',),
)
```

Initializes a `WoodwindFingering` from another `WoodwindFingering`:

```
>>> woodwind_fingering_2 = instrumenttools.WoodwindFingering(
...     woodwind_fingering)
>>> print format(woodwind_fingering_2)
instrumenttools.WoodwindFingering(
    instrument_name='clarinet',
    center_column=('one', 'two', 'three', 'five'),
    left_hand=('R', 'thumb'),
    right_hand=('e',),
)
```

Calls a `WoodwindFingering` to create a woodwind diagram `MarkupCommand`:

```
>>> fingering_command = woodwind_fingering()
>>> print format(fingering_command)
markuptools.MarkupCommand(
    'woodwind-diagram',
    schemetools.Scheme(
        'clarinet'
```

```

    ),
    schemetools.Scheme(
        schemetools.SchemePair(
            'cc',
            ('one', 'two', 'three', 'five')
        ),
        schemetools.SchemePair(
            'lh',
            ('R', 'thumb')
        ),
        schemetools.SchemePair(
            'rh',
            ('e',)
        )
    )
)

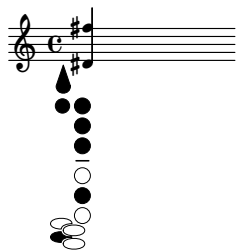
```

Attaches 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")
>>> attach(markup, chord)
>>> show(chord)

```



Initializes 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(instrument_name='piccolo', center_column=(), left_hand=(), right_hand=())
WoodwindFingering(instrument_name='flute', center_column=(), left_hand=(), right_hand=())
WoodwindFingering(instrument_name='oboe', center_column=(), left_hand=(), right_hand=())
WoodwindFingering(instrument_name='clarinet', center_column=(), left_hand=(), right_hand=())
WoodwindFingering(instrument_name='bass-clarinet', center_column=(), left_hand=(), right_hand=())
WoodwindFingering(instrument_name='saxophone', center_column=(), left_hand=(), right_hand=())
WoodwindFingering(instrument_name='bassoon', center_column=(), left_hand=(), right_hand=())
WoodwindFingering(instrument_name='contrabassoon', center_column=(), left_hand=(), right_hand=())

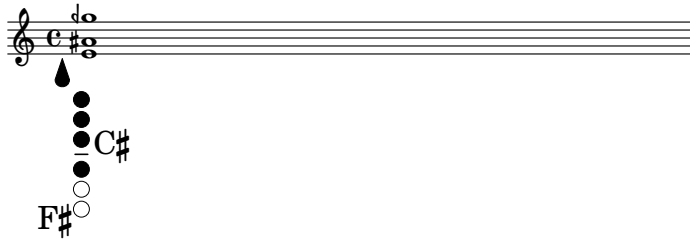
```

An override displays diagrams symbolically instead of graphically:

```

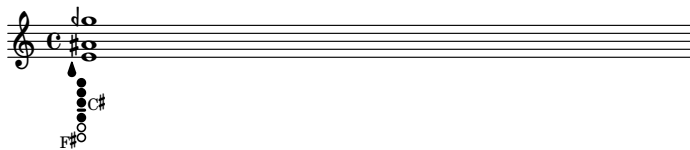
>>> chord = Chord("<e' as' gqf''>1")
>>> fingering = instrumenttools.WoodwindFingering(
...     'clarinet',
...     center_column=['one', 'two', 'three', 'four'],
...     left_hand=['R', 'cis'],
...     right_hand=['fis'])
>>> diagram = fingering()
>>> not_graphical = markuptools.MarkupCommand(
...     'override',
...     schemetools.SchemePair('graphical', False))
>>> markup = markuptools.Markup(
...     [not_graphical, diagram], direction=Down)
>>> attach(markup, chord)
>>> show(chord)

```



The thickness and size of diagrams can also be changed with overrides:

```
>>> chord = Chord("<e' as' ggf'>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)
>>> attach(markup, 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.

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__**()

Calls woodwind fingering.

Returns markup command.

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

WoodwindFingering.**__format__**(*format_specification*='')

Formats woodwind fingering.

Set *format_specification* to '' or 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

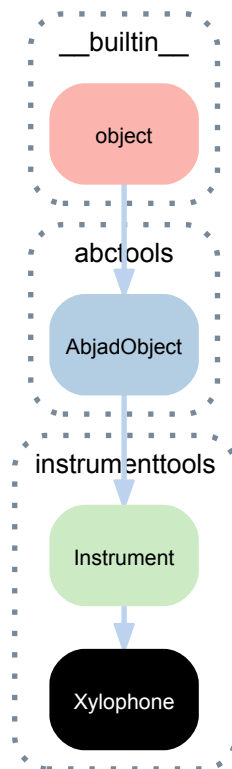
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

5.1.53 instrumenttools.Xylophone



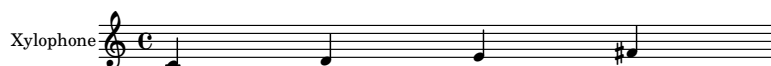
```

class instrumenttools.Xylophone (instrument_name='xylophone',
                                short_instrument_name='xyl.',          in-
                                strument_name_markup=None,
                                short_instrument_name_markup=None,     allow-
                                able_clefs=None,      pitch_range='[C4,   C7]', sound-
                                ing_pitch_of_written_middle_c='C5')
  
```

A xylphone.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> xylophone = instrumenttools.Xylophone()
>>> attach(xylophone, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `__builtin__.object`

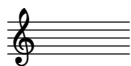
Read-only properties

`Xylophone.allowable_clefs`
Gets xylophone's allowable clefs.

```

>>> xylophone.allowable_clefs
ClefInventory([Clef('treble')])
  
```

```
>>> show(xylophone.allowable_clefs)
```



Returns clef inventory.

Xylophone.instrument_name

Gets xylophone's name.

```
>>> xylophone.instrument_name
'xylophone'
```

Returns string.

Xylophone.instrument_name_markup

Gets xylophone's instrument name markup.

```
>>> xylophone.instrument_name_markup
Markup(('Xylophone',))
```

```
>>> show(xylophone.instrument_name_markup)
```

Xylophone

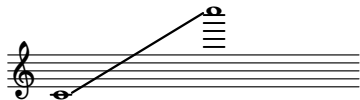
Returns markup.

Xylophone.pitch_range

Gets xylophone's range.

```
>>> xylophone.pitch_range
PitchRange('C4, C7')
```

```
>>> show(xylophone.pitch_range)
```



Returns pitch range.

Xylophone.short_instrument_name

Gets xylophone's short instrument name.

```
>>> xylophone.short_instrument_name
'xyl.'
```

Returns string.

Xylophone.short_instrument_name_markup

Gets xylophone's short instrument name markup.

```
>>> xylophone.short_instrument_name_markup
Markup(('Xyl.',))
```

```
>>> show(xylophone.short_instrument_name_markup)
```

Xyl.

Returns markup.

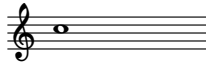
Xylophone.sounding_pitch_of_written_middle_c

Gets sounding pitch of xylophone's written middle C.

```
>>> xylophone.sounding_pitch_of_written_middle_c
NamedPitch('c''')
```



```
>>> show(xylophone.sounding_pitch_of_written_middle_c)
```



Returns named pitch.

Special methods

(Instrument) .**__copy__**(**args*)

Copies instrument.

Returns new instrument.

(Instrument) .**__eq__**(*arg*)

Is true when *arg* is an instrument with instrument name and short instrument name equal to those of this instrument. Otherwise false.

Returns boolean.

(Instrument) .**__format__**(*format_specification*='')

Formats instrument.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Instrument) .**__hash__**()

Gets hash value instrument.

Computed on type, instrument name and short instrument name.

Returns integer.

(Instrument) .**__makenew__**(*instrument_name=None, short_instrument_name=None, instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range=None, sounding_pitch_of_written_middle_c=None*)

Makes new instrument.

Returns new instrument.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Instrument) .**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.2 Functions

5.2.1 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")
>>> violin = instrumenttools.Violin()
>>> attach(violin, staff)
```

```
>>> list(
... instrumenttools.iterate_out_of_range_notes_and_chords(
... staff))
[Chord('<d fs>8')]
```

Returns generator.

5.2.2 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")
>>> violin = instrumenttools.Violin()
>>> attach(violin, staff)
```

```
>>> instrumenttools.notes_and_chords_are_in_range(staff)
True
```

Otherwise false:

```
>>> staff = Staff("c'8 r8 <d fs>8 r8")
>>> violin = instrumenttools.Violin()
>>> attach(violin, staff)
```

```
>>> instrumenttools.notes_and_chords_are_in_range(staff)
False
```

Returns boolean.

5.2.3 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")
>>> clef = Clef('treble')
>>> attach(clef, staff)
>>> violin = instrumenttools.Violin()
>>> attach(violin, staff)
```

```
>>> instrumenttools.notes_and_chords_are_on_expected_clefs(staff)
True
```

Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> clef = Clef('alto')
>>> attach(clef, staff)
>>> violin = instrumenttools.Violin()
>>> attach(violin, staff)
```

```
>>> 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")
>>> clef = Clef('percussion')
>>> attach(clef, staff)
>>> violin = instrumenttools.Violin()
>>> attach(violin, staff)
```

```
>>> 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.

5.2.4 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")
>>> clarinet = instrumenttools.ClarinetInBFlat()
>>> attach(clarinet, staff)
>>> show(staff)
```



```
>>> instrumenttools.transpose_from_sounding_pitch_to_written_pitch(staff)
>>> show(staff)
```



Returns none.

5.2.5 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.ClarinetInBFlat()
>>> attach(clarinet, staff)
>>> show(staff)
```



```
>>> instrumenttools.transpose_from_written_pitch_to_sounding_pitch(staff)
>>> show(staff)
```



Returns none.

6.1 Functions

6.1.1 `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*.

6.1.2 `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.

6.1.3 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*.

6.1.4 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.

6.1.5 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.

6.1.6 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')`

Colors measures with non-power-of-two denominators in *expr* with *color*.

```
>>> staff = Staff(Measure((2, 8), "c'8 d'8") * 2)
>>> scoretools.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.

6.1.7 `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*.

6.1.8 `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")
>>> scoretools.FixedDurationTuplet(Duration(2, 8), staff[-3:])
FixedDurationTuplet(Duration(1, 4), "e'8 f'8 g'8")
>>> labeltools.label_leaves_in_expr_with_leaf_depth(staff)
>>> show(staff)
```



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



Returns none.

6.1.9 `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 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.

6.1.10 `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 = scoretools.RhythmicStaff([tuplet])
>>> override(staff).text_script.staff_padding = 2.5
>>> override(staff).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 = scoretools.RhythmicStaff([tuplet])
>>> override(staff).text_script.staff_padding = 2.5
>>> override(staff).time_signature.stencil = False
>>> labeltools.label_leaves_in_expr_with_leaf_durations(
...     tuplet,
...     label_durations=True,
...     label_written_durations=False)
```

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

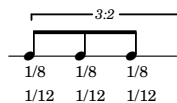


Example 3. Label leaves in tuplet with both written and actual durations:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> staff = scoretools.RhythmicStaff([tuplet])
>>> override(staff).text_script.staff_padding = 2.5
>>> override(staff).time_signature.stencil = False
>>> labeltools.label_leaves_in_expr_with_leaf_durations(
...     tuplet,
```

```
...     label_durations=True,
...     label_written_durations=True)
```

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



Returns none.

6.1.11 `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)
>>> print format(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.

6.1.12 `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)
>>> print format(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.

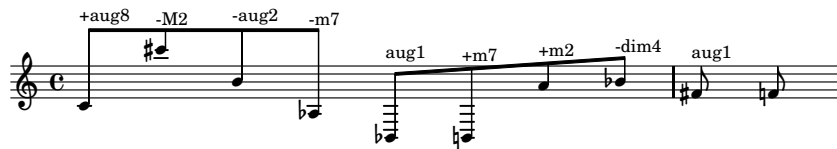
6.1.13 `labeltools.label_leaves_in_expr_with_named_interval_classes`

`labeltools.label_leaves_in_expr_with_named_interval_classes` (*expr*,
markup_direction=Up)

Label leaves in *expr* with named interval classes:

```
>>> notes = scoretools.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.

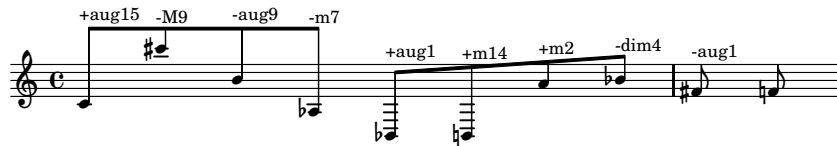
6.1.14 `labeltools.label_leaves_in_expr_with_named_intervals`

`labeltools.label_leaves_in_expr_with_named_intervals` (*expr*,
markup_direction=Up)

Label leaves in *expr* with named intervals:

```
>>> notes = scoretools.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.

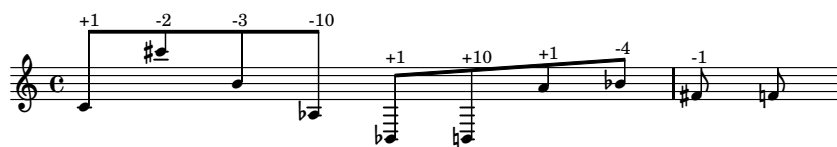
6.1.15 `labeltools.label_leaves_in_expr_with_numbered_interval_classes`

`labeltools.label_leaves_in_expr_with_numbered_interval_classes` (*expr*,
markup_direction=Up)

Label leaves in *expr* with numbered interval classes:

```
>>> notes = scoretools.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.

6.1.16 `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 = scoretools.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.

6.1.17 `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 = scoretools.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.

6.1.18 `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 format(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)
```



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 format(staff)
\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.

6.1.19 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(scoretools.make_leaves([None, 12, [13, 14, 15], None], [(1, 4)]))
>>> labeltools.label_leaves_in_expr_with_pitch_numbers(staff)
>>> print format(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.

6.1.20 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")
>>> scoretools.FixedDurationTuplet(Duration(2, 8), staff[-3:])
FixedDurationTuplet(Duration(1, 4), "e'8 f'8 g'8")
>>> labeltools.label_leaves_in_expr_with_tuplet_depth(staff)
>>> show(staff)
```



```
>>> print format(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.

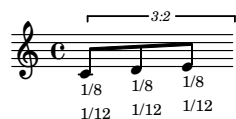
6.1.21 labeltools.label_leaves_in_expr_with_written_leaf_duration

`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_leaf_durations(tuplet)
>>> print format(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.

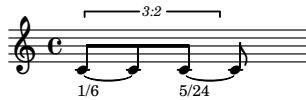
6.1.22 labeltools.label_logical_ties_in_expr_with_logical_tie_duration

`labeltools.label_logical_ties_in_expr_with_logical_tie_duration` (*expr*,
markup_direction=Down)

Label logical ties in *expr* with logical tie durations:

```
>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_logical_ties_in_expr_with_logical_tie_duration(staff)
```

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



Returns none.

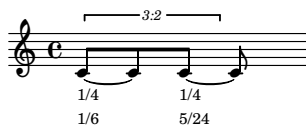
6.1.23 `labeltools.label_logical_ties_in_expr_with_logical_tie_durations`

`labeltools.label_logical_ties_in_expr_with_logical_tie_durations` (*expr*,
markup_direction=Down)

Label logical ties in *expr* with both written logical tie duration and logical tie duration:

```
>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_logical_ties_in_expr_with_logical_tie_durations(staff)
```

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



Returns none.

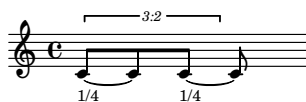
6.1.24 `labeltools.label_logical_ties_in_expr_with_written_logical_tie_duration`

`labeltools.label_logical_ties_in_expr_with_written_logical_tie_duration` (*expr*,
markup_direction=Down)

Label logical ties in *expr* with written logical tie duration:

```
>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_logical_ties_in_expr_with_written_logical_tie_duration(
...     staff)
```

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



Returns none.

6.1.25 `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.

6.1.26 `labeltools.label_vertical_moments_in_expr_with_interval_class_vectors`

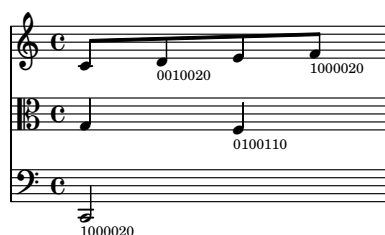
`labeltools.label_vertical_moments_in_expr_with_interval_class_vectors` (*expr*,
markup_direction=Down)

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.

6.1.27 `labeltools.label_vertical_moments_in_expr_with_named_intervals`

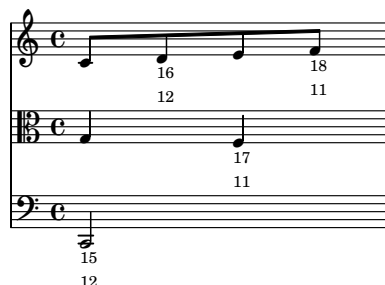
`labeltools.label_vertical_moments_in_expr_with_named_intervals` (*expr*,
markup_direction=Down)

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.

6.1.28 `labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes`

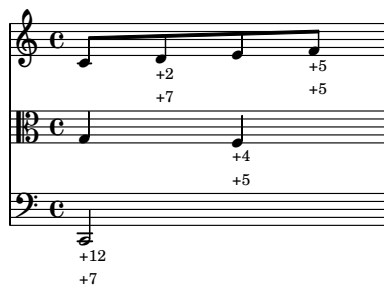
`labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes` (*expr*,
markup_direction=Down)

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.

6.1.29 `labeltools.label_vertical_moments_in_expr_with_numbered_intervals`

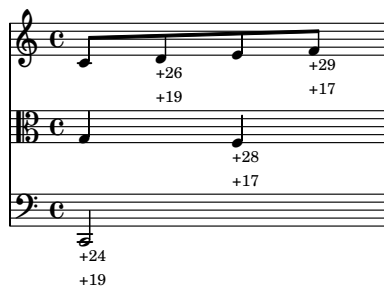
`labeltools.label_vertical_moments_in_expr_with_numbered_intervals` (*expr*,
markup_direction=Down)

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.

6.1.30 `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.

6.1.31 `labeltools.label_vertical_moments_in_expr_with_pitch_numbers`

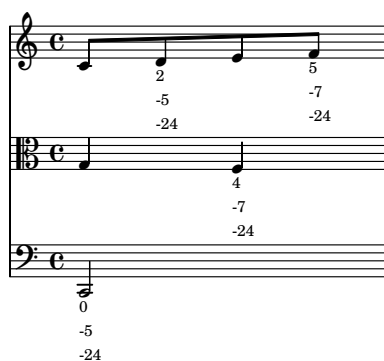
`labeltools.label_vertical_moments_in_expr_with_pitch_numbers` (*expr*,
markup_direction=Down)

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.

6.1.32 `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)
>>> print format(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)
>>> print format(staff)
\new Staff {
  c'8
  d'8
  e'8
  f'8
}
```

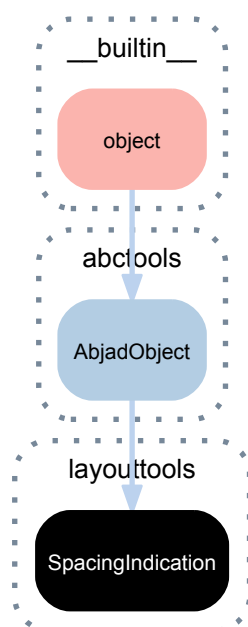
```
>>> show(staff)
```



Returns none.

7.1 Concrete classes

7.1.1 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 and proportional notation duration:

```
>>> tempo = Tempo(Duration(1, 8), 44)
>>> indication = layouttools.SpacingIndication(tempo, Duration(1, 68))
```

```
>>> indication
SpacingIndication(Tempo(Duration(1, 8), 44), Duration(1, 68))
```

Initialize from constants:

```
>>> layouttools.SpacingIndication((1, 8), 44, (1, 68))
SpacingIndication(Tempo(Duration(1, 8), 44), Duration(1, 68))
```

Initialize from other spacing indication:

```
>>> layouttools.SpacingIndication(indication)
SpacingIndication(Tempo(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 contextualize.

`SpacingIndication.tempo_indication`
Abjad tempo indication object.

Special methods

`SpacingIndication.__eq__(expr)`
Spacing indications compare equal when normalized spacing durations compare equal.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

7.2 Functions

7.2.1 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.

7.2.2 layouttools.set_line_breaks_by_line_duration

```
layouttools.set_line_breaks_by_line_duration(expr, line_duration,
                                             line_break_class=None,
                                             kind='prolated',
                                             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.

7.2.3 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,
                                                           add_empty_bars=False)
```

Iterate *line_break_class* instances in *expr* and accumulate 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)
```



```
>>> print format(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.

7.2.4 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,
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 = Tempo(Duration(1, 8), 44)
>>> attach(tempo, staff, scope=Staff)
>>> show(staff)
```



```
>>> layouttools.set_line_breaks_cyclically_by_line_duration_in_seconds_ge(
...     staff, Duration(6))
>>> show(staff)
```

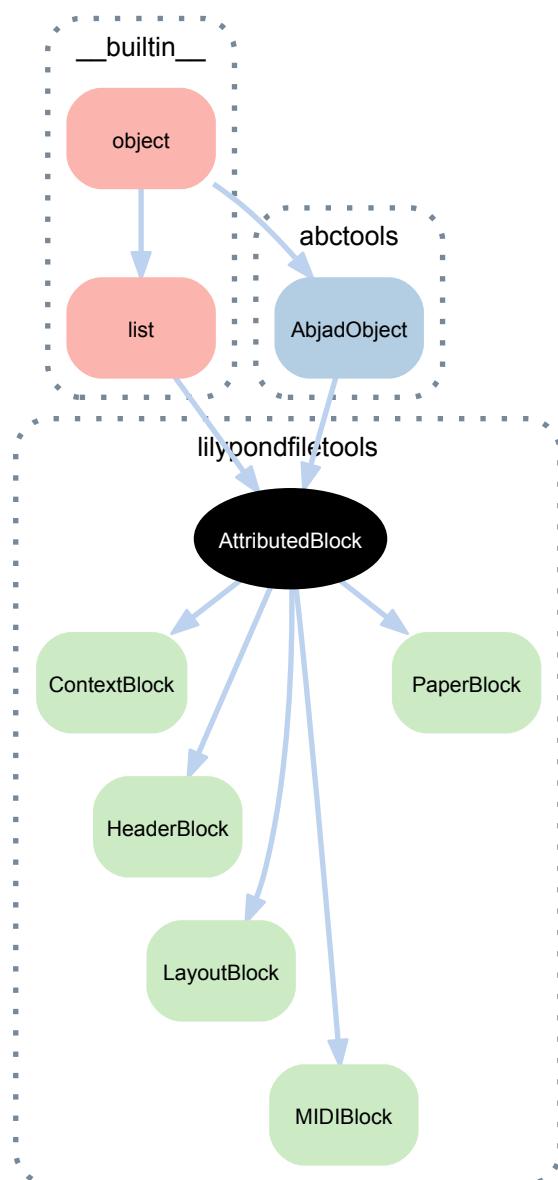


```
>>> print format(staff)
\new Staff {
  \tempo 8=44
  {
    \time 2/8
    c'8
    d'8
  }
  {
    e'8
    f'8
    \break
  }
  {
    g'8
    a'8
  }
  {
    b'8
    c''8
  }
}
```

When `line_break_class=None` set `line_break_class` to measure.

8.1 Abstract classes

8.1.1 lilypondfiletools.AttributedBlock



class lilypondfiletools.**AttributedBlock**
Abjad model of LilyPond input file block with attributes.

Bases

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

Read/write properties

`AttributedBlock.is_formatted_when_empty`

True when attributed block is formatted when empty. Otherwise false.

Returns boolean.

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`

`AttributedBlock.__format__ (format_specification='')`

Formats attributed block.

Returns string.

`(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__ ()`

Gets interpreter representation of attributed block.

Returns string.

`(list) .__reversed__ ()`

`L.__reversed__()` – return a reverse iterator over the list

`(list) .__rmul__ ()`

`x.__rmul__(n) <==> n*x`

`(list) .__setitem__ ()`

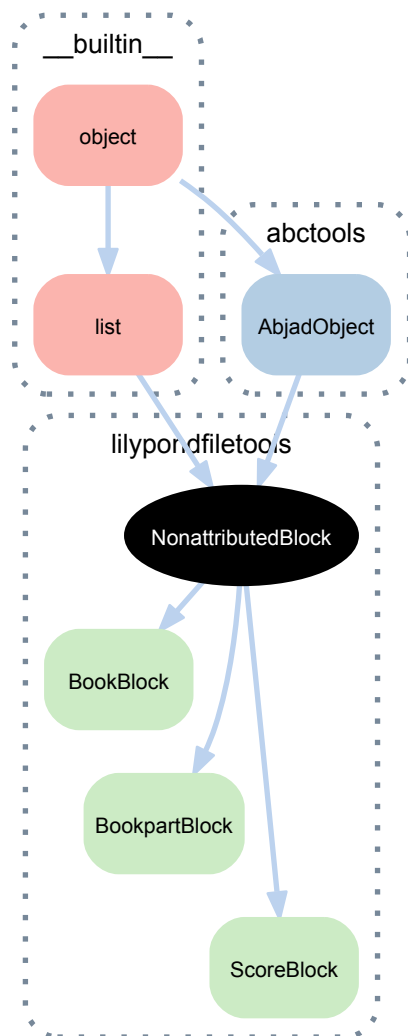
`x.__setitem__(i, y) <==> x[i]=y`

`(list) .__setslice__ ()`

`x.__setslice__(i, j, y) <==> x[i:j]=y`

Use of negative indices is not supported.

8.1.2 lilypondfiletools.NonattributedBlock



class `lilypondfiletools.NonattributedBlock`
 Abjad model of LilyPond input file block with no attributes.

Bases

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

Read/write properties

`NonattributedBlock.is_formatted_when_empty`
 True when nonattributed block is formatted when empty. Otherwise false.
 Returns boolean.

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

NonattributedBlock.**__format__**(format_specification='')
Formats nonattributed block.

Returns string.

(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

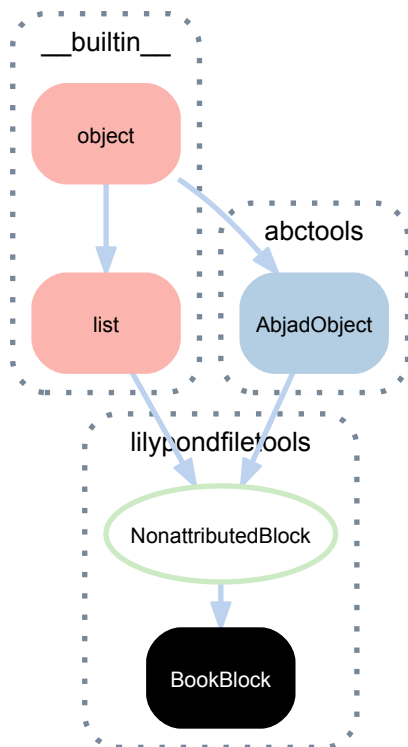
(list).**__imul__**()
x.__imul__(y) <==> x*=y

(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__()
    Gets interpreter representation of nonattributed block.
    Returns string.
(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.
```

8.2 Concrete classes

8.2.1 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/write properties

`(NonattributedBlock).is_formatted_when_empty`
 True when nonattributed block is formatted when empty. Otherwise false.
 Returns boolean.

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
```

```
(NonattributedBlock) .__format__(format_specification='')
    Formats nonattributed block.

    Returns string.
```

```
(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__()
    Gets interpreter representation of nonattributed block.

    Returns string.
```

```
(list) .__reversed__()
    L.__reversed__() – return a reverse iterator over the list
```

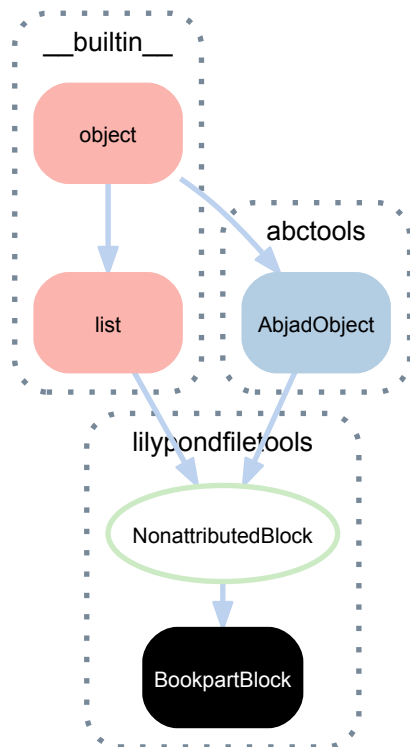
```
(list) .__rmul__()
    x.__rmul__(n) <==> n*x
```

```
(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.
```


8.2.2 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/write properties

`(NonattributedBlock).is_formatted_when_empty`
 True when nonattributed block is formatted when empty. Otherwise false.
 Returns boolean.

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

(NonattributedBlock).**__format__**(format_specification='')
 Formats nonattributed block.
 Returns string.

(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

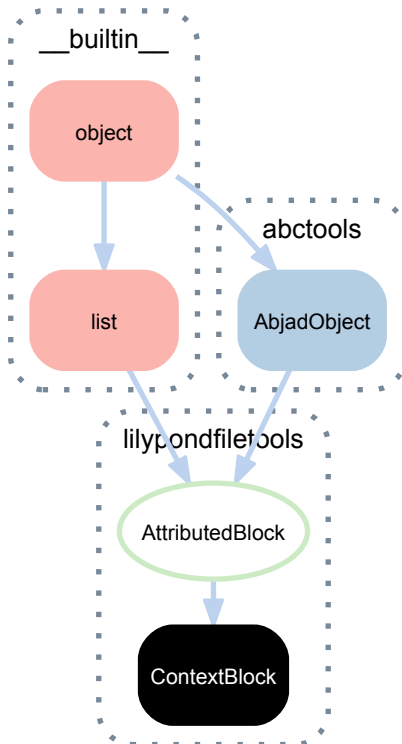
(list).**__iter__**() <==> iter(x)

```

(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__()
    Gets interpreter representation of nonattributed block.
    Returns string.
(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.

```

8.2.3 lilypondfiletools.ContextBlock



class lilypondfiletools.**ContextBlock** (*context_name=None*)
 A LilyPond input file context block.

```
>>> context_block = lilypondfiletools.ContextBlock()
```

```
>>> context_block
ContextBlock()
```

```
>>> context_block.context_name = 'Score'
>>> override(context_block).bar_number.transparent = True
>>> scheme = schemetools.Scheme('end-of-line-invisible')
>>> override(context_block).time_signature.break_visibility = scheme
>>> moment = schemetools.SchemeMoment((1, 45))
>>> contextualize(context_block).proportionalNotationDuration = moment
```

Bases

- `lilypondfiletools.AttributedBlock`
- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ContextBlock.accepts`

Context accepts commands.

Returns sets.

`ContextBlock.engraver_consists`

Engraver consists commands.

Returns set.

`ContextBlock.engraver_removals`

Engraver removal commands.

Returns set.

Read/write properties

`ContextBlock.alias`

Gets and sets alias of context block.

Returns string or none.

`ContextBlock.context_name`

Gets and sets context name of context block.

Returns string or none.

`(AttributedBlock).is_formatted_when_empty`

True when attributed block is formatted when empty. Otherwise false.

Returns boolean.

`ContextBlock.name`

Gets and sets name of context block.

Returns string or none.

`ContextBlock.type`

Gets and sets LilyPond type of context block.

Returns string or none.

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

(AttributedBlock) **.__format__** (format_specification='')
 Formats attributed block.
 Returns string.

(list) **.__ge__()**
 x.__ge__(y) <==> x>=y

(list) **.__getitem__()**
 x.__getitem__(y) <==> x[y]

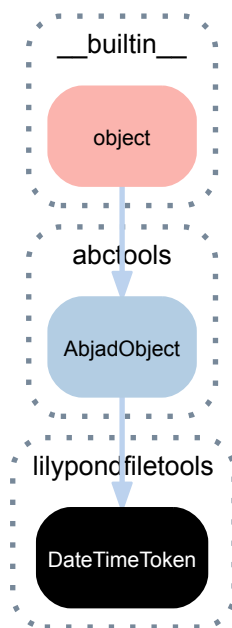
(list) **.__getslice__()**
 x.__getslice__(i, j) <==> x[i:j]
 Use of negative indices is not supported.

(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__()
    Gets interpreter representation of attributed block.
    Returns string.
(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.
```

8.2.4 lilypondfiletools.DateToken



class lilypondfiletools.DateToken
 Date / time token.

```
>>> lilypondfiletools.DateTimeToken()
DateTimeToken(...)
```

Returns date / time token.

Bases

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

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`DateTimeToken.__format__(format_specification='')`

Gets format.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

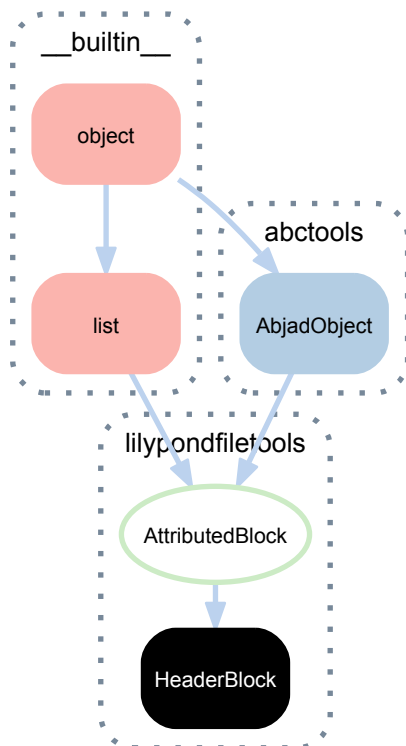
Returns boolean.

`DateTimeToken.__repr__()`

Gets interpreter representation of date / time token.

Returns string.

8.2.5 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/write properties

(`AttributedBlock`).**`is_formatted_when_empty`**

True when attributed block is formatted when empty. Otherwise false.

Returns boolean.

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

(AttributedBlock) .__format__(format_specification='')
    Formats attributed block.

    Returns string.

(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__()
    Gets interpreter representation of attributed block.

    Returns string.

(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

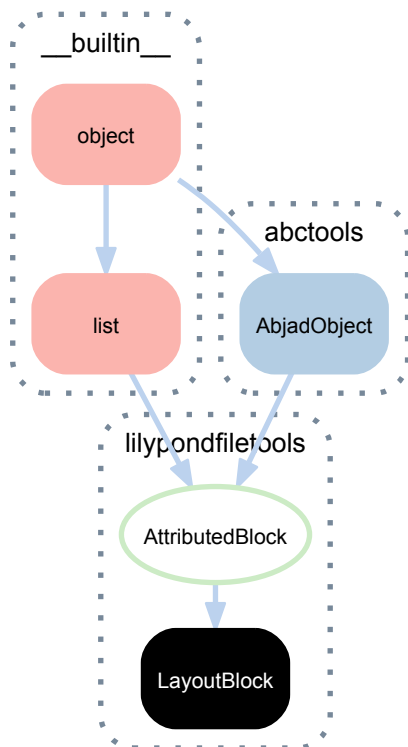
```

```
(list).__setslice__()
```

`x.__setslice__(i, j, y) <==> x[i:j]=y`

Use of negative indices is not supported.

8.2.6 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')
>>> override(context_block).bar_number.transparent = True

>>> scheme_expr = schemetools.Scheme('end-of-line-invisible')
>>> override(context_block).time_signature.break_visibility = \
...     scheme_expr
>>> layout_block.context_blocks.append(context_block)

```

Returns list.

`LayoutBlock.contexts`

DEPRECATED. USE `CONTEXT_BLOCKS` INSTEAD.

Read/write properties

`(AttributedBlock).is_formatted_when_empty`

True when attributed block is formatted when empty. Otherwise false.

Returns boolean.

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

(AttributedBlock) .__format__ (format_specification='')
    Formats attributed block.

    Returns string.

(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__()
    Gets interpreter representation of attributed block.

    Returns string.

(list) .__reversed__()
    L.__reversed__() – return a reverse iterator over the list

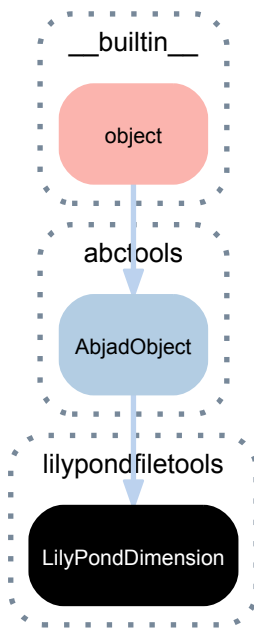
(list) .__rmul__()
    x.__rmul__(n) <==> n*x

(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.
```

8.2.7 lilypondfiletools.LilyPondDimension



class `lilypondfiletools.LilyPondDimension` (*value=0, unit='cm'*)
 A LilyPond page dimension.

```
>>> dimension = lilypondfiletools.LilyPondDimension(2, 'in')
```

Bases

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

Read-only properties

`LilyPondDimension.unit`
 Unit of LilyPond dimension.
 Returns string.

`LilyPondDimension.value`
 Value of LilyPond dimension.
 Returns float.

Special methods

`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`LilyPondDimension.__format__(format_specification='')`
 Formats LilyPond dimension.
 Returns string.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.

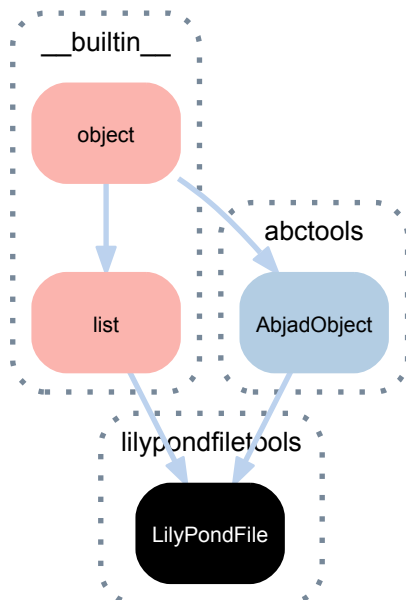
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

8.2.8 lilypondfiletools.LilyPondFile



class lilypondfiletools.**LilyPondFile**

A 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/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.

(AbjadObject).**__eq__**(*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

LilyPondFile.**__format__**(*format_specification*='')
 Formats LilyPond file.
 Returns string.

(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

LilyPondFile.**__illustrate__**()
 Illustrates LilyPond file.
 Returns LilyPond file unchanged.

(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

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

LilyPondFile.**__repr__**()
 Gets interpreter representation of LilyPond file.
 Returns string.

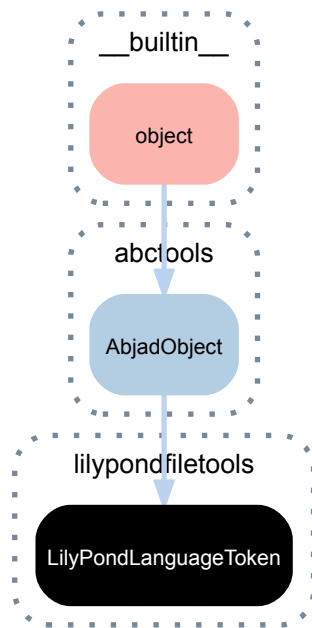
(list).**__reversed__**()
 L.**__reversed__**() – return a reverse iterator over the list

(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.

8.2.9 lilypondfiletools.LilyPondLanguageToken



class lilypondfiletools.LilyPondLanguageToken
LilyPond language token.

```
>>> lilypondfiletools.LilyPondLanguageToken()
LilyPondLanguageToken('english')
```

Bases

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

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`LilyPondLanguageToken.__format__(format_specification='')`

Formats LilyPond language token.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

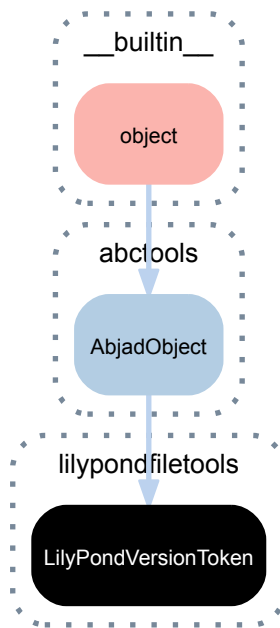
Returns boolean.

`LilyPondLanguageToken.__repr__()`

Gets interpreter representation of LilyPond language token.

Returns string.

8.2.10 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.version`

Version of LilyPond version token.

Returns string.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`LilyPondVersionToken.__format__(format_specification='')`

Formats LilyPond version token.

Return string.

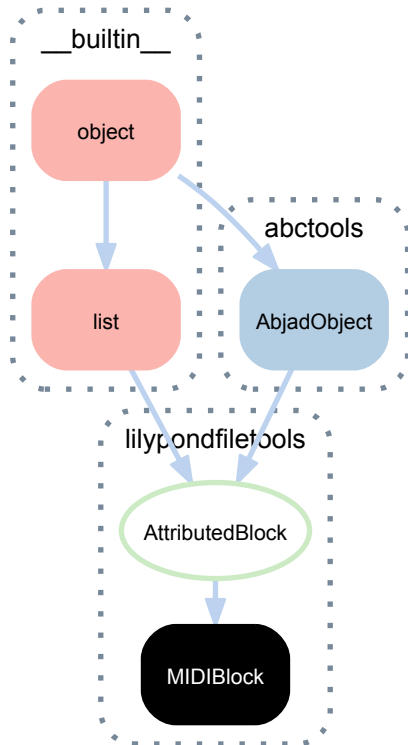
`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`LilyPondVersionToken.__repr__()`
 Gets interpreter representation of LilyPond version token.
 Returns string.

8.2.11 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/write properties

(AttributedBlock).**is_formatted_when_empty**
 True when attributed block is formatted when empty. Otherwise false.
 Returns boolean.

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

(AttributedBlock).**__format__**(format_specification='')
 Formats attributed block.
 Returns string.

(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__()
    Gets interpreter representation of attributed block.

    Returns string.
```

```
(list).__reversed__()
    L.__reversed__() – return a reverse iterator over the list
```

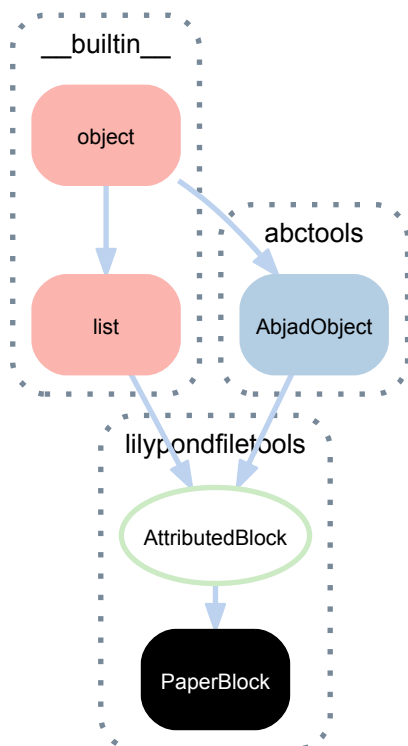
```
(list).__rmul__()
    x.__rmul__(n) <==> n*x
```

```
(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.
```

8.2.12 lilypondfiletools.PaperBlock



class lilypondfiletools.**PaperBlock**

A 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
```

Bases

- lilypondfiletools.AttributedBlock
- __builtin__.list
- abctools.AbjadObject
- __builtin__.object

Read/write properties

(AttributedBlock).**is_formatted_when_empty**

True when attributed block is formatted when empty. Otherwise false.

Returns boolean.

PaperBlock.**minimal_page_breaking**

Gets and sets minimal page-breaking flag on paper block.

Returns boolean.

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

(AttributedBlock) **.__format__** (format_specification='')
 Formats attributed block.
 Returns string.

(list) **.__ge__()**
 x.__ge__(y) <==> x>=y

(list) **.__getitem__()**
 x.__getitem__(y) <==> x[y]

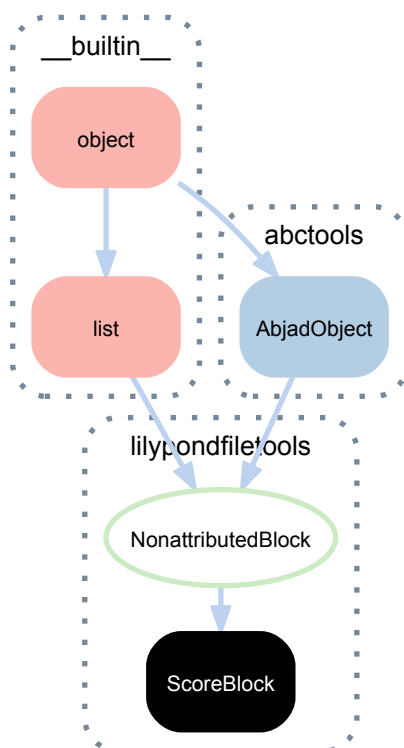
(list) **.__getslice__()**
 x.__getslice__(i, j) <==> x[i:j]
 Use of negative indices is not supported.

(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__()
    Gets interpreter representation of attributed block.
    Returns string.
(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.
```

8.2.13 lilypondfiletools.ScoreBlock



class `lilypondfiletools.ScoreBlock`

Abjad model of LilyPond input file score block:

```
>>> score_block = lilypondfiletools.ScoreBlock()
```

```
>>> score_block
ScoreBlock()
```

```
>>> score_block.append(Staff([]))
>>> print format(score_block)
\score {
  \new Staff {
  }
}
```

ScoreBlocks does not format when empty, as this generates a parser error in LilyPond:

```
>>> score_block = lilypondfiletools.ScoreBlock()
>>> format(score_block) == ''
True
```

Returns score block.

Bases

- `lilypondfiletools.NonattributedBlock`
- `__builtin__.list`
- `abctools.AbjadObject`
- `__builtin__.object`

Read/write properties

(`NonattributedBlock`).**is_formatted_when_empty**

True when nonattributed block is formatted when empty. Otherwise false.

Returns boolean.

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

(NonattributedBlock).**__format__**(format_specification='')
Formats nonattributed block.
Returns string.

(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__**()
Gets interpreter representation of nonattributed block.
Returns string.

```
(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.
```

8.3 Functions

8.3.1 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.

8.3.2 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.

8.3.3 lilypondfiletools.make_time_signature_context_block

`lilypondfiletools.make_time_signature_context_block` (*font_size=3, minimum_distance=12, padding=4*)

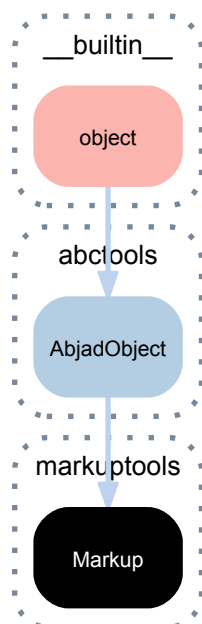
Make time signature context block:

```
>>> context_block = lilypondfiletools.make_time_signature_context_block()
```

Returns context block.

9.1 Concrete classes

9.1.1 markuptools.Markup



class `markuptools.Markup` (*argument*, *direction=None*, *markup_name=None*)
Abjad model of LilyPond markup.

Initializes from string:

```
>>> markup = markuptools.Markup(r'\bold { "This is markup text." }')
```

```
>>> markup
Markup((MarkupCommand('bold', ['This is markup text.']),))
```

```
>>> show(markup)
```

This is markup text.

Initializes any markup from existing markup:

```
>>> markup_1 = markuptools.Markup('foo', direction=Up)
>>> markup_2 = markuptools.Markup(markup_1, direction=Down)
```

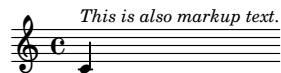
Attach markup to score components by calling them on the component:

```
>>> note = Note("c'4")
```

```
>>> markup = markuptools.Markup(
...     r'\italic { "This is also markup text." }', direction=Up)
```

```
>>> attach(markup, note)
```

```
>>> show(note)
```



Set *direction* to Up, Down, 'neutral', '^', '_', '-' or None.

Markup objects are immutable.

Returns markup instance.

Bases

- `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.direction`

Direction of markup.

Returns ordinal constant.

`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.

Special methods

`Markup.__copy__(*args)`

Copies markup.

Returns new markup.

`Markup.__eq__(expr)`

True when *expr* is a markup with format equal to that of this markup. Otherwise false.

Returns boolean.

`Markup.__format__(format_specification='')`
 Formats markup.
 Set *format_specification* to `'`, `'lilypond'` or `'storage'`. Interprets `"` equal to `'lilypond'`.
 Returns string.

`Markup.__hash__()`
 Hashes markup.
 Returns integer.

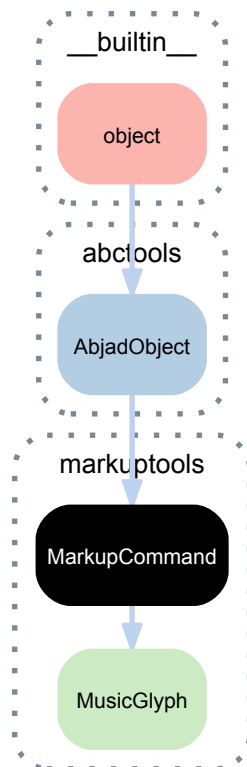
`Markup.__illustrate__()`
 Illustrates markup.
 Returns LilyPond file.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

`Markup.__str__()`
 String representation of markup.
 Returns string.

9.1.2 markuptools.MarkupCommand



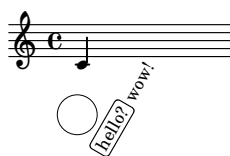
`class markuptools.MarkupCommand(command=None, *args)`
 A LilyPond markup command.

```
>>> 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 format(combine, 'lilypond')
\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)
>>> attach(markup, note)
>>> show(note)
```



Markup commands are immutable.

Bases

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

Read-only properties

`MarkupCommand.args`

Markup command arguments.

Returns tuple.

`MarkupCommand.command`

Markup command name.

Returns string.

Special methods

`MarkupCommand.__eq__ (expr)`

True when *expr* is a markup command with command and args equal to those of this markup command. Otherwise false.

Returns boolean.

`MarkupCommand.__format__ (format_specification='')`

Formats markup command.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`. Interprets `"` equal to `'storage'`.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

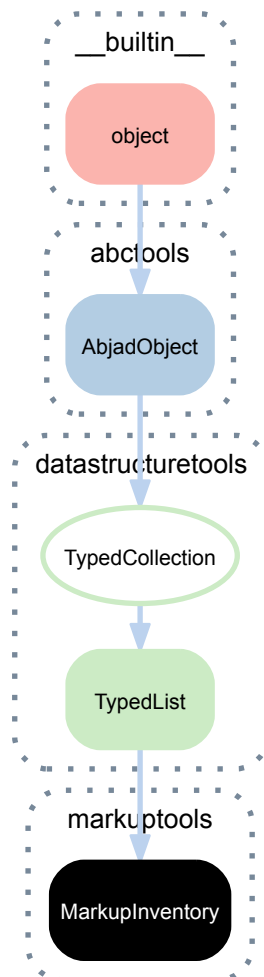
Returns string.

MarkupCommand.**__str__**()

String representation of markup command.

Returns string.

9.1.3 markuptools.MarkupInventory



class markuptools.**MarkupInventory** (*tokens=None, item_class=None, keep_sorted=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.

Returns string or none.

`(TypedList).keep_sorted`
Sorts collection on mutation if true.

Methods

`(TypedList).append(token)`
Changes *token* to item and appends.

```
>>> 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)`
Changes *token* to item and returns 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)`
Changes *tokens* to items and extends.

```
>>> 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*)

Changes *token* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *token*)

Changes *token* to item and inserts.

```
>>> 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.

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

Aliases list.pop().

(TypedList) **.remove** (*token*)

Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) .**__format__** (*format_specification*='')

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList) .**__getitem__** (*i*)

Aliases list.**__getitem__**().

(TypedList) .**__iadd__** (*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList (
    [
        indicatortools.Dynamic(
            'ppp'
        ),
        indicatortools.Dynamic(
            'p'
        ),
        indicatortools.Dynamic(
            'mp'
        ),
        indicatortools.Dynamic(
            'mf'
        ),
        indicatortools.Dynamic(
            'fff'
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection) .**__iter__** ()

Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__** ()

Length of typed collection.

Returns nonnegative integer.

(TypedList) .**__makenew__** (*tokens=None, item_class=None, keep_sorted=None, cus-*
tom_identifier=None)

Makes new typed list.

Returns new typed list.

(TypedCollection) .**__ne__** (*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList) .**__reversed__** ()

Aliases list.**__reversed__**().

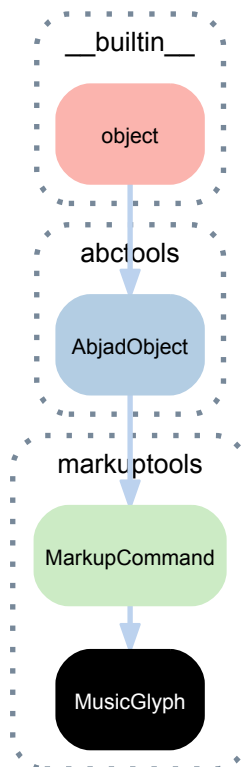
(TypedList).**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```

9.1.4 markuptools.MusicGlyph



class markuptools.**MusicGlyph** (*glyph_name=None*)
A LilyPond music glyph.

```
>>> markuptools.MusicGlyph('accidentals.sharp')
MusicGlyph('accidentals.sharp')
```

```
>>> print _  
\musicglyph #"accidentals.sharp"
```

Bases

- `markuptools.MarkupCommand`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(MarkupCommand) **.args**
Markup command arguments.

Returns tuple.

(MarkupCommand) **.command**
Markup command name.

Returns string.

Special methods

(MarkupCommand) **.__eq__** (*expr*)
True when *expr* is a markup command with command and args equal to those of this markup command.
Otherwise false.

Returns boolean.

(MarkupCommand) **.__format__** (*format_specification*='')
Formats markup command.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`. Interprets `"` equal to `'storage'`.

Returns string.

(AbjadObject) **.__ne__** (*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject) **.__repr__** ()
Gets interpreter representation of Abjad object.

Returns string.

(MarkupCommand) **.__str__** ()
String representation of markup command.

Returns string.

9.2 Functions

9.2.1 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 = format(markup, 'lilypond')

>>> 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.

9.2.2 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 format(markup, 'lilypond')
\markup {
  \fill-line
  {
    \bold
    \fontsize
    #3
    \concat
    {
      \on-the-fly
      #print-page-number-check-first
      \fromproperty
      #'page:page-number-string
    }
  }
}
```

Returns markup.

9.2.3 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.

9.2.4 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 format(markup, 'lilypond')
\markup {
  \override
    #'(font-name . "Times")
    \fontsize
      #18
    \column
      {
        \center-align
          {
            {
              \vspace
                #6
              \line
                {
                  "String Quartet"
                }
              \vspace
                #12
            }
          }
        }
      }
}
```

Returns markup.

9.2.5 `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:

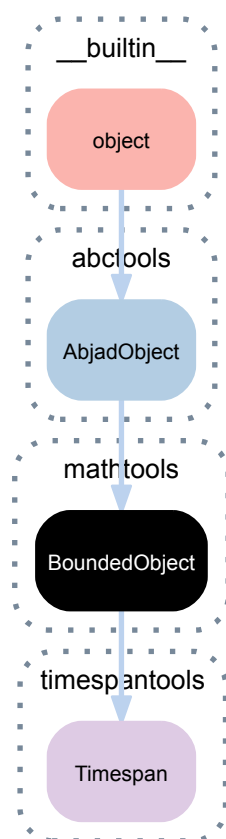
```
>>> markup = markuptools.make_vertically_adjusted_composer_markup('Josquin Desprez')
```

```
>>> print format(markup, 'lilypond')
\markup {
  \override
    #'(font-name . "Times")
    {
      \hspace
        #0
      \raise
        #-20
      \fontsize
        #3
        "Josquin Desprez"
      \hspace
        #0
    }
}
```

Returns markup.

10.1 Concrete classes

10.1.1 `mathtools.BoundedObject`



class `mathtools.BoundedObject`
Bounded object mix-in.

Bases

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

Read-only properties

`BoundedObject.is_closed`

True when left closed and right closed. Otherwise false.

Returns boolean.

`BoundedObject.is_half_closed`

True when left closed xor right closed.

Returns boolean.

`BoundedObject.is_half_open`

True when left and right open are not the same. Otherwise false.

Return boolean.

`BoundedObject.is_open`

True when left or right open. Otherwise false.

Returns boolean.

Read/write properties

`BoundedObject.is_left_closed`

True when left closed. Otherwise false.

Returns boolean.

`BoundedObject.is_left_open`

True when left open. Otherwise false.

Returns boolean.

`BoundedObject.is_right_closed`

True when right closed. Otherwise false.

Returns boolean.

`BoundedObject.is_right_open`

True when right open. Otherwise false.

Returns boolean.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

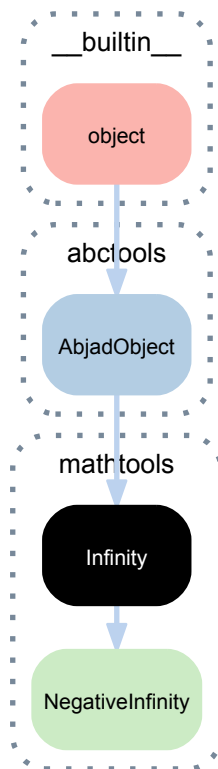
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

10.1.2 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)`

True when *expr* is also infinity. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

Infinity.**__ge__**(*expr*)

True for all values of *expr*. Otherwise false.

Returns boolean.

Infinity.**__gt__**(*expr*)

True for all noninfinite values of *expr*. Otherwise false.

Returns boolean.

Infinity.**__le__**(*expr*)

True when *expr* is infinite. Otherwise false.

Returns boolean.

Infinity.**__lt__**(*expr*)

True for no values of *expr*.

Returns boolean.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

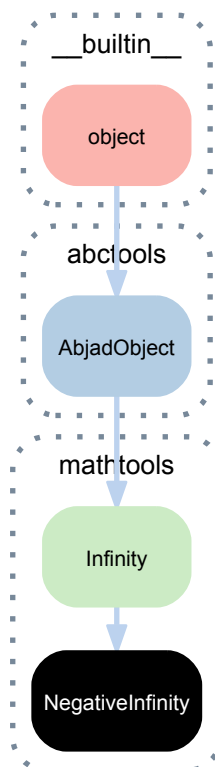
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

10.1.3 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)`True when *expr* is also infinity. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Infinity).__ge__(expr)`True for all values of *expr*. Otherwise false.

Returns boolean.

`(Infinity).__gt__(expr)`True for all noninfinite values of *expr*. Otherwise false.

Returns boolean.

`(Infinity).__le__(expr)`True when *expr* is infinite. Otherwise false.

Returns boolean.

`(Infinity).__lt__(expr)`True for no values of *expr*.

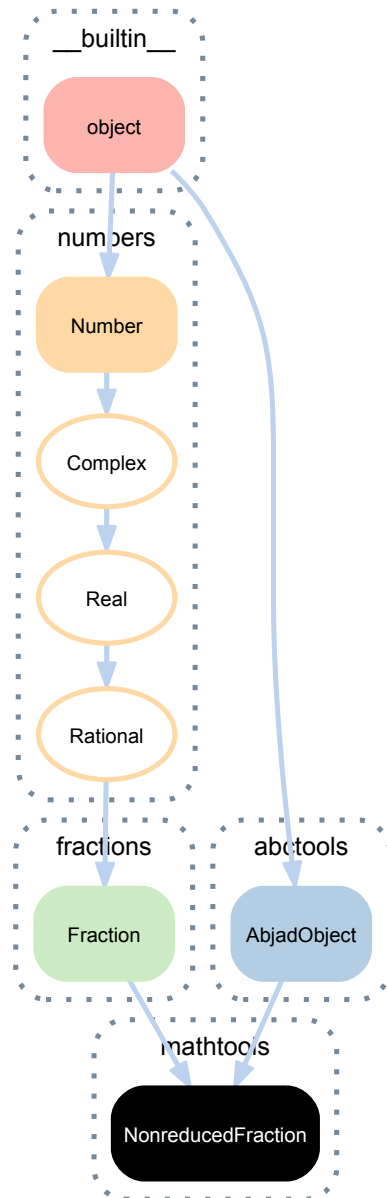
Returns boolean.

`(AbjadObject).__ne__(expr)`Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

10.1.4 `mathtools.NonreducedFraction`



class `mathtools.NonreducedFraction`
 Initializes with an integer numerator and integer denominator:

```
>>> mathtools.NonreducedFraction(3, 6)
NonreducedFraction(3, 6)
```

Initializes with only an integer denominator:

```
>>> mathtools.NonreducedFraction(3)
NonreducedFraction(3, 1)
```

Initializes with an integer pair:

```
>>> mathtools.NonreducedFraction((3, 6))
NonreducedFraction(3, 6)
```

Initializes 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
>>> systemtools.IOManager.count_function_calls(
...     'fractions.Fraction(3, 6)', globals())
13
```

```
>>> systemtools.IOManager.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.

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)
>>> systemtools.IOManager.count_function_calls('a + b', globals())
67
```

Adding an integer is even faster:

```
>>> systemtools.IOManager.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.__format__(format_specification='')
Formats nonreduced fraction.
```

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
>>> print format(fraction)
mathtools.NonreducedFraction(-6, 3)
```

Returns string.

```
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.

`NonreducedFraction.__repr__()`
Gets interpreter representation of nonreduced fraction.

```
>>> mathtools.NonreducedFraction(3, 6)
NonreducedFraction(3, 6)
```

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.

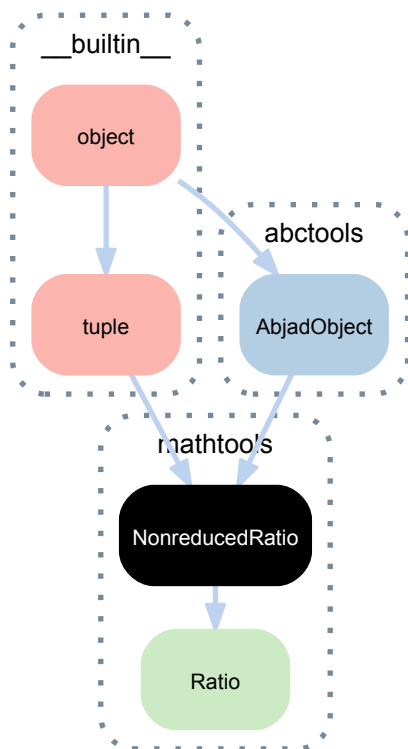
```
>>> mathtools.NonreducedFraction(3, 3) - 2
NonreducedFraction(-3, 3)
```

Returns nonreduced fraction.

`(Fraction).__truediv__(a, b)`
`a / b`

`(Fraction).__trunc__(a)`
`trunc(a)`

10.1.5 `mathtools.NonreducedRatio`



class `mathtools.NonreducedRatio`

Nonreduced ratio of one or more nonzero integers.

Initializes from one or more nonzero integers:

```
>>> mathtools.NonreducedRatio(2, 4, 2)
NonreducedRatio(2, 4, 2)
```

Initializes from a tuple or list:

```
>>> ratio = mathtools.NonreducedRatio((2, 4, 2))
>>> ratio
NonreducedRatio(2, 4, 2)
```

Uses 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__**()
 $x.__add__(y) \iff x+y$

(tuple).**__contains__**()
 $x.__contains__(y) \iff y \text{ in } x$

NonreducedRatio.**__eq__**(*expr*)
 True when *expr* is a nonreduced ratio with numerator and denominator equal to those of this nonreduced ratio. Otherwise false.

Returns boolean.

NonreducedRatio.**__format__**(*format_specification*='')
 Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(tuple).**__ge__**()
 $x.__ge__(y) \iff x \geq y$

(tuple).**__getitem__**()
 $x.__getitem__(y) \iff x[y]$

(tuple).**__getslice__**()
 $x.__getslice__(i, j) \iff x[i:j]$

Use of negative indices is not supported.

(tuple).**__gt__**()
 $x.__gt__(y) \iff x > y$

(tuple).**__hash__**() $\iff \text{hash}(x)$

(tuple).**__iter__**() $\iff \text{iter}(x)$

(tuple).**__le__**()
 $x.__le__(y) \iff x \leq y$

(tuple).**__len__**() $\iff \text{len}(x)$

(tuple).**__lt__**()
 $x.__lt__(y) \iff x < y$

(tuple).**__mul__**()
 $x.__mul__(n) \iff x*n$

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

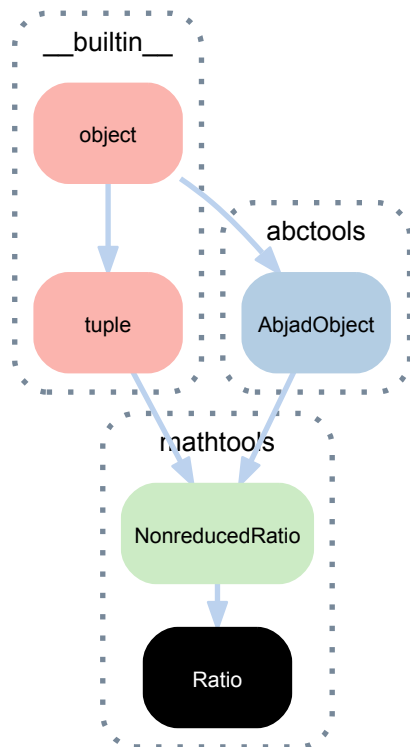
NonreducedRatio.**__new__**(*args)

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.

Returns string.

(tuple).**__rmul__**()
 $x.__rmul__(n) \iff n*x$

10.1.6 mathtools.Ratio



class `mathtools.Ratio`

Ratio of one or more nonzero integers.

Initializes from one or more nonzero integers:

```
>>> mathtools.Ratio(2, 4, 2)
Ratio(1, 2, 1)
```

Initializes from a tuple or list:

```
>>> ratio = mathtools.Ratio((2, 4, 2))
>>> ratio
Ratio(1, 2, 1)
```

Uses 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)
 True when *expr* is a nonreduced ratio with numerator and denominator equal to those of this nonreduced ratio. Otherwise false.

Returns boolean.

(NonreducedRatio).**__format__**(format_specification='')
 Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(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)
 Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

Ratio.**__new__**(*args)

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.

Returns string.

(tuple).**__rmul__**()
 x.__rmul__(n) <==> n*x

`Ratio.__str__()`

String representation of ratio.

Returns string.

10.2 Functions

10.2.1 `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.

10.2.2 `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.

10.2.3 `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.

10.2.4 `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.

10.2.5 `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.

10.2.6 `mathtools.cumulative_sums`

`mathtools.cumulative_sums(sequence, start=0)`

Cumulative sums of *sequence*.

```

>>> mathtools.cumulative_sums([1, 2, 3, 4, 5, 6, 7, 8], start=None)
[1, 3, 6, 10, 15, 21, 28, 36]

```

Returns list.

10.2.7 `mathtools.cumulative_sums_pairwise`

`mathtools.cumulative_sums_pairwise(sequence)`

Lists pairwise cumulative sums of *sequence* from 0.

```

>>> mathtools.cumulative_sums_pairwise([1, 2, 3, 4, 5, 6])
[(0, 1), (1, 3), (3, 6), (6, 10), (10, 15), (15, 21)]

```

Returns list of pairs.

10.2.8 `mathtools.difference_series`

`mathtools.difference_series(sequence)`
Difference series of *sequence*.

```
>>> mathtools.difference_series([1, 1, 2, 3, 5, 5, 6])  
[0, 1, 1, 2, 0, 1]
```

Returns list.

10.2.9 `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.

10.2.10 `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:

```
>>> systemtools.IOManager.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.

10.2.11 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.

10.2.12 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.

10.2.13 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.

10.2.14 `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.

10.2.15 `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.

10.2.16 `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.

10.2.17 `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.

10.2.18 `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.

10.2.19 `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.

10.2.20 `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.

10.2.21 `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 $0 \leq j$, $2 < k$.

10.2.22 `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.

10.2.23 `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.

10.2.24 `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.

10.2.25 `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.

10.2.26 `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.

10.2.27 `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.

10.2.28 `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.

10.2.29 `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.

10.2.30 `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.

10.2.31 `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.

10.2.32 `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.

10.2.33 `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.

10.2.34 `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.

10.2.35 `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.

10.2.36 `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.

10.2.37 `mathtools.partition_integer_into_halves`

`mathtools.partition_integer_into_halves` (n , *bigger*='left', *even*='allowed')

Writes positive integer n as the pair $t = (\text{left}, \text{right})$ such that $n == \text{left} + \text{right}$.

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.

10.2.38 `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 .

```
>>> 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.

10.2.39 `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.

10.2.40 `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.

10.2.41 `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.

10.2.42 `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.

10.2.43 `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.

10.2.44 `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.

10.2.45 `mathtools.yield_nonreduced_fractions`

`mathtools.yield_nonreduced_fractions()`

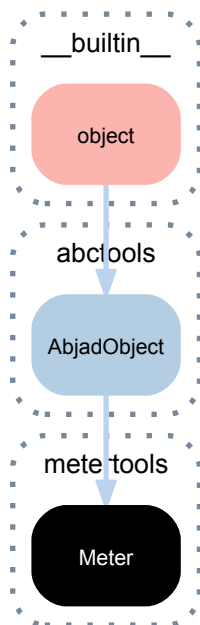
Yields positive nonreduced fractions in Cantor diagonalized order:

```
>>> 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.

11.1 Concrete classes

11.1.1 `metertools.Meter`



class `metertools.Meter` (*arg=None, 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 meter models many parts of the common practice understanding of meter:

```
>>> meter = metertools.Meter((4, 4))
```

```
>>> meter
Meter('(4/4 (1/4 1/4 1/4 1/4))')
```

```
>>> print meter.pretty_rtm_format
(4/4 (
  1/4
  1/4
  1/4
  1/4))
```

```
>>> meter = metertools.Meter((3, 4))
>>> print meter.pretty_rtm_format
(3/4 (
  1/4
  1/4
  1/4))
```

```
>>> meter = metertools.Meter((6, 8))
>>> print meter.pretty_rtm_format
(6/8 (
  (3/8 (
    1/8
    1/8
    1/8))
  (3/8 (
    1/8
    1/8
    1/8))))
```

```
>>> meter = metertools.Meter((5, 4))
>>> print meter.pretty_rtm_format
(5/4 (
  (3/4 (
    1/4
    1/4
    1/4))
  (2/4 (
    1/4
    1/4))))
```

```
>>> meter = metertools.Meter(
...     (5, 4), decrease_durations_monotonically=False)
>>> print meter.pretty_rtm_format
(5/4 (
  (2/4 (
    1/4
    1/4))
  (3/4 (
    1/4
    1/4
    1/4))))
```

```
>>> meter = metertools.Meter((12, 8))
>>> print meter.pretty_rtm_format
(12/8 (
  (3/8 (
    1/8
    1/8
    1/8))
  (3/8 (
    1/8
    1/8
    1/8))
  (3/8 (
    1/8
    1/8
    1/8))
  (3/8 (
    1/8
    1/8
    1/8))))
```

Returns meter object.

Bases

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

Read-only properties

`Meter.decrease_durations_monotonically`

True if the meter divides large primes into collections of 2 and 3 that decrease monotonically.

Example 1. Metrical hierarchy with durations that increase monotonically:

```
>>> meter = \
...     metertools.Meter((5, 4),
...     decrease_durations_monotonically=False)
```

```
>>> meter.decrease_durations_monotonically
False
```

```
>>> print meter.pretty_rtm_format
(5/4 (
  (2/4 (
    1/4
    1/4))
  (3/4 (
    1/4
    1/4
    1/4))))
```

Example 2. Meter with durations that decrease monotonically:

```
>>> meter = \
...     metertools.Meter((5, 4),
...     decrease_durations_monotonically=True)
```

```
>>> meter.decrease_durations_monotonically
True
```

```
>>> print meter.pretty_rtm_format
(5/4 (
  (3/4 (
    1/4
    1/4
    1/4))
  (2/4 (
    1/4
    1/4))))
```

Returns boolean.

`Meter.denominator`

Beat hierarchy denominator:

```
>>> meter.denominator
4
```

Returns positive integer.

`Meter.depthwise_offset_inventory`

Depthwise inventory of offsets at each grouping level:

```
>>> for depth, offsets in enumerate(
...     meter.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.

`Meter.graphviz_format`

Graphviz format of hierarchy's root node:

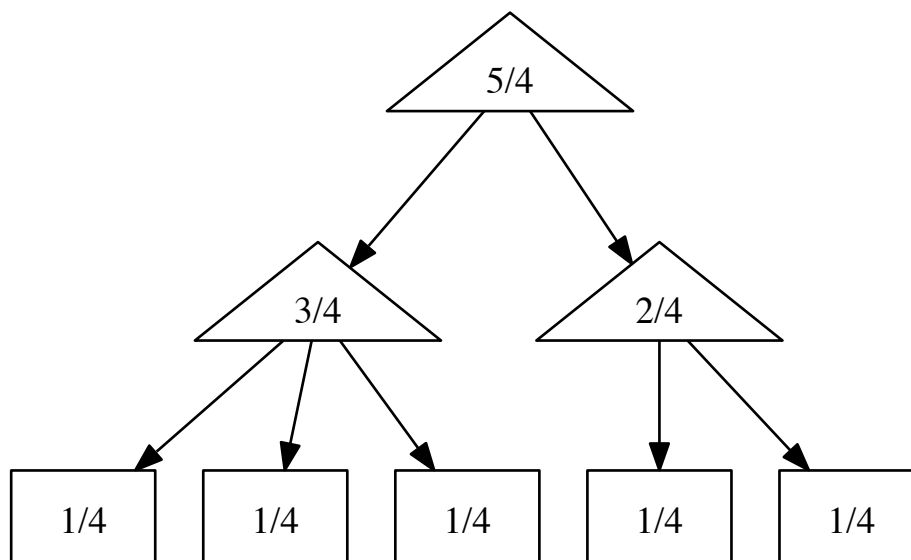
```
>>> print meter.graphviz_format
digraph G {
```

```

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;
}

```

```
>>> topleveltools.graph(meter)
```



Returns string.

Meter.implied_time_signature

Implied time signature:

```

>>> metertools.Meter((4, 4)).implied_time_signature
TimeSignature((4, 4))

```

Returns TimeSignature object.

Meter.numerator

Beat hierarchy numerator:

```

>>> meter.numerator
5

```

Returns positive integer.

Meter.preprolated_duration

Beat hierarchy preprolated_duration:

```
>>> meter.preprolated_duration
Duration(5, 4)
```

Returns `preprolated_duration`.

Meter.pretty_rtm_format

Beat hierarchy pretty RTM format:

```
>>> print meter.pretty_rtm_format
(5/4 (
  (3/4 (
    1/4
    1/4
    1/4))
  (2/4 (
    1/4
    1/4))))
```

Returns string.

Meter.root_node

Beat hierarchy root node:

```
>>> meter.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.

Meter.rtm_format

Beat hierarchy RTM format:

```
>>> meter.rtm_format
'(5/4 ((3/4 (1/4 1/4 1/4)) (2/4 (1/4 1/4))))'
```

Returns string.

Methods

`Meter.generate_offset_kernel_to_denominator` (*denominator*, *normalize=True*)

Generate a dictionary of all offsets in a meter up to *denominator*, where the keys are the offsets and the values are the normalized weights of those offsets:

```
>>> meter = \
...     metertools.Meter((4, 4))
>>> kernel = \
...     meter.generate_offset_kernel_to_denominator(8)
>>> for offset, weight in sorted(kernel.kernel.iteritems()):
...     print '{!s}\t{!s}'.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 meter.

Returns dictionary.

Static methods

`Meter.fit_meters_to_expr` (*expr*, *meters*, *discard_final_orphan_downbeat=True*, *starting_offset=None*, *denominator=32*)

Find the best-matching sequence of meters for the offsets contained in *expr*.

```
>>> meters = [metertools.Meter(x)
...           for x in [(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 metertools.Meter.fit_meters_to_expr(
...     expr, meters):
...     print x.implicit_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 metertools.Meter.fit_meters_to_expr(
...     expr, meters):
...     print x.implicit_time_signature
...
5/4
5/4
5/4
5/4
```

Offsets are coerced from *expr* via *MetricAccentKernel.count_offsets_in_expr()*.

MetricalHierarchies are coerced from *meters* via *MetricalHierarchyInventory*.

Returns list.

Special methods

`Meter.__eq__(expr)`

True when *expr* is a meter with an rtm format equal to that of this meter. Otherwise false.

Returns boolean.

`Meter.__format__(format_specification='')`

Formats meter.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

```
>>> meter = metertools.Meter((5, 4))
>>> print format(meter)
metertools.Meter(
    ' (5/4 ((3/4 (1/4 1/4 1/4)) (2/4 (1/4 1/4)))) '
)
```

Returns string.

`Meter.__iter__()`

Iterates meter.

```
>>> meter = metertools.Meter((5, 4))
```

```
>>> for x in meter:
...     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))
```

Yields pairs.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

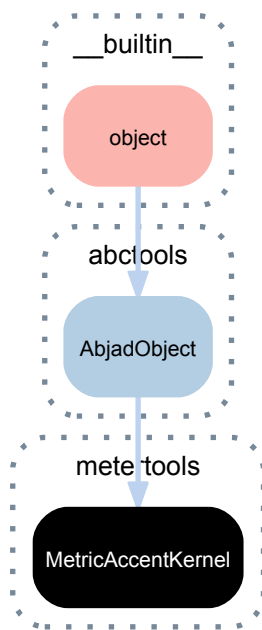
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

11.1.2 metertools.MetricAccentKernel



class metertools.MetricAccentKernel (*kernel=None*)
 A metrical kernel, or offset-impulse-response-filter.

```

>>> hierarchy = metertools.Meter((5, 8))
>>> kernel = hierarchy.generate_offset_kernel_to_denominator(8)
>>> kernel
MetricAccentKernel(
  {
    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 *MetricAccentKernel* instance.

Bases

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

Read-only properties

`MetricAccentKernel.kernel`

The kernel datastructure.

Returns dict.

Static methods

`MetricAccentKernel.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)
```



```
>>> MetricAccentKernel = metertools.MetricAccentKernel
>>> leaves = score.select_leaves(
...     allow_discontiguous_leaves=True)
>>> counter = MetricAccentKernel.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 = MetricAccentKernel.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

`MetricAccentKernel.__call__(expr)`

Calls metrical accent kernel on *expr*.

Returns float.

`MetricAccentKernel.__eq__(expr)`

True when *expr* is a metrical accent kernel with a kernel equal to that of this metrical accent kernel. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

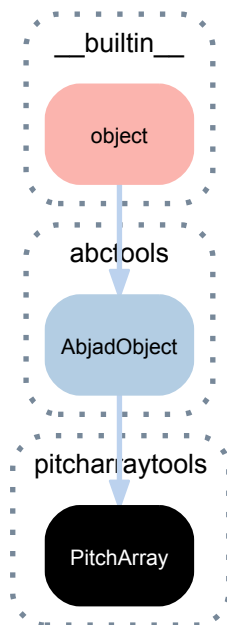
Gets interpreter representation of Abjad object.

Returns string.

PITCHARRAYTOOLS

12.1 Concrete classes

12.1.1 `pitcharraytools.PitchArray`



class `pitcharraytools.PitchArray` (*args)
A two-dimensional array of pitches.

Bases

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

Read-only properties

`PitchArray.cell_tokens_by_row`
Cells tokens of pitch array by row.
Returns tuple.

`PitchArray.cell_widths_by_row`
Cell widths of pitch array by row.
Returns tuple.

`PitchArray.cells`

Cells of pitch array.

Returns set.

`PitchArray.columns`

Columns of pitch array.

Returns tuple.

`PitchArray.depth`

Depth of pitch array.

Defined equal to number of pitch array rows in pitch array.

Returns nonnegative integer.

`PitchArray.dimensions`

Dimensions of pitch array.

Returns pair.

`PitchArray.has_voice_crossing`

True when pitch array has voice crossing. Otherwise false.

Returns boolean.

`PitchArray.is_rectangular`

True when no rows in pitch array are defective. Otherwise false.

Returns boolean.

`PitchArray.pitches`

Pitches in pitch array.

Returns tuple.

`PitchArray.pitches_by_row`

Pitches in pitch array by row.

Returns tuple.

`PitchArray.rows`

Rows in pitch array.

Returns tuple.

`PitchArray.size`

Size of pitch array.

Defined equal to the product of depth and width.

Returns nonnegative integer.

`PitchArray.voice_crossing_count`

Voice crossing count.

Returns nonnegative integer.

`PitchArray.weight`

Weight of pitch array.

Defined equal to the sum of the weight of the rows in pitch array.

Returns nonnegative integer.

`PitchArray.width`

Width of pitch array.

Defined equal to the width of the widest row in pitch array.

Returns nonnegative integer.

Methods

`PitchArray.append_column(column)`

Append *column* to pitch array.

Returns none.

`PitchArray.append_row(row)`

Appends *row* to pitch array.

Returns none.

`PitchArray.apply_pitches_by_row(pitch_lists)`

Applies *pitch_lists* to pitch array by row.

Returns none.

`PitchArray.copy_subarray(upper_left_pair, lower_right_pair)`

Copies subarray of pitch array.

Returns new pitch array.

`PitchArray.has_spanning_cell_over_index(index)`

True when pitch array has one or more spanning cells over *index*. Otherwise false.

Returns boolean.

`PitchArray.list_nonspanning_subarrays()`

Lists 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)`

Pads pitch array to *depth*.

Returns none.

`PitchArray.pad_to_width(width)`

Pads pitch array to *width*.

Returns none.

`PitchArray.pop_column(column_index)`
 Pops column *column_index* from pitch array.
 Returns pitch array column.

`PitchArray.pop_row(row_index=-1)`
 Pops row *row_index* from pitch array.
 Returns pitch array row.

`PitchArray.remove_row(row)`
 Removes *row* from pitch array.
 Returns none.

`PitchArray.to_measures(cell_duration_denominator=8)`
 Changes 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.

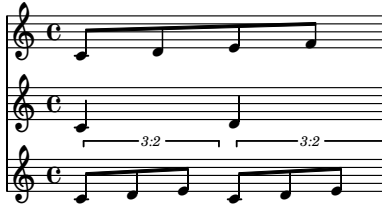
Static methods

`PitchArray.from_score(score, populate=True)`
 Makes pitch array from *score*.

Example 1. Makes 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(
...         scoretools.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. Makes 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(
...         scoretools.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)`

Concatenates *arg* to pitch array.

Returns new pitch array.

`PitchArray.__contains__(arg)`

True when pitch array contains *arg*. Otherwise false.

Returns boolean.

`PitchArray.__copy__()`

Copies pitch array.

Returns new pitch array.

`PitchArray.__eq__(arg)`

True when *arg* is a pitch array with contents equal to that of this pitch array. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

PitchArray.**__getitem__**(*arg*)

Gets row *arg* from pitch array.

Returns pitch array row.

PitchArray.**__iadd__**(*arg*)

Adds *arg* to pitch array in place.

```
>>> 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 pitch array.

PitchArray.**__ne__**(*arg*)

True when pitch array does not equal *arg*. Otherwise false.

Returns boolean.

PitchArray.**__repr__**()

Intepreter representation of pitch array.

Returns string.

PitchArray.**__setitem__**(*i*, *arg*)

Sets pitch array row *i* to *arg*.

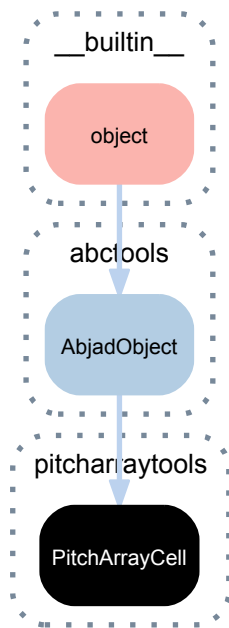
Retunrs none.

PitchArray.**__str__**()

String representation of pitch array.

Returns string.

12.1.2 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.previous
PitchArrayCell(x1)
```

```
>>> cell.row_index
0
```

```
>>> cell.token
2
```

```
>>> cell.width
2
```

Bases

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

Read-only properties

`PitchArrayCell.column_indices`

Tuple of one or more nonnegative integer indices.

Returns tuple.

`PitchArrayCell.indices`

Indices of pitch array cell.

Returns pair.

`PitchArrayCell.is_first_in_row`

True when pitch array cell is first in row. Otherwise false.

Returns boolean.

`PitchArrayCell.is_last_in_row`

True when pitch array cell is last in row. Otherwise false.

Returns boolean.

`PitchArrayCell.next`

Gets next pitch array cell in row after this pitch array cell.

Returns pitch array cell.

`PitchArrayCell.parent_array`

Gets pitch array that houses pitch array cell.

Return pitch array.

`PitchArrayCell.parent_column`

Gets column that houses pitch array cell.

Returns pitch array column.

`PitchArrayCell.parent_row`

Gets pitch array rown that houses pitch array cell.

Returns pitch array row.

`PitchArrayCell.previous`

Gets pitch array cell in row prior to this pitch array cell.

Returns pitch arracy cell.

`PitchArrayCell.row_index`

Row index of pitch array cell.

Returns nonnegative integer or none.

`PitchArrayCell.token`

Token of pitch array cell.

`PitchArrayCell.weight`

Weight of pitch array cell.

Defined equal to number of pitches in pitch array cell.

Returns nonnegative integer.

`PitchArrayCell.width`

Width of pitch array cell.

Returns positive integer.

Read/write properties

`PitchArrayCell.pitches`

Gets and sets pitches of pitch array cell.

Returns list.

Methods

`PitchArrayCell.matches_cell(arg)`

True when pitch array cell matches *arg*. Otherwise false.

Returns boolean.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`PitchArrayCell.__repr__()`

Gets interpreter representation of pitch array cell.

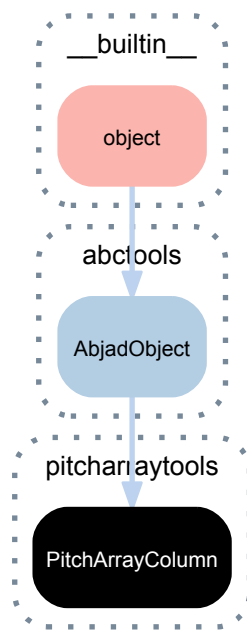
Returns string.

`PitchArrayCell.__str__()`

String representation of pitch array cell.

Returns string.

12.1.3 `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'    ]
```

Bases

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

Read-only properties

`PitchArrayColumn.cell_tokens`
 Cells tokens of pitch array column.
 Returns tuple.

`PitchArrayColumn.cell_widths`
 Cell widths of pitch array column.
 Returns tuple.

`PitchArrayColumn.cells`
 Cells of pitch array column.
 Returns tuple.

`PitchArrayColumn.column_index`

Column index of pitch array column.

Returns nonnegative integer.

`PitchArrayColumn.depth`

Depth of pitch array column.

Defined equal to number of pitch array cells in pitch array column.

Returns nonnegative integer.

`PitchArrayColumn.dimensions`

Dimensions of pitch array column.

Returns pair.

`PitchArrayColumn.has_voice_crossing`

True when pitch array column has voice crossing. Otherwise false.

Returns boolean.

`PitchArrayColumn.is_defective`

True when pitch array column depth does not equal depth of parent array. Otherwise false.

Returns boolean.

`PitchArrayColumn.parent_array`

Parent array that houses pitch array column.

Returns pitch array.

`PitchArrayColumn.pitches`

Pitches in pitch array column.

Returns tuple.

`PitchArrayColumn.start_cells`

Start cells in pitch array column.

Returns tuple.

`PitchArrayColumn.start_pitches`

Start pitches in pitch array column.

Returns tuple.

`PitchArrayColumn.stop_cells`

Stop cells in pitch array column.

Returns tuple.

`PitchArrayColumn.stop_pitches`

Stop pitches in pitch array column.

Returns tuple.

`PitchArrayColumn.weight`

Weight of pitch array column.

Defined equal to the sum of the weight of pitch array cells in pitch array column.

Returns nonnegative integer.

`PitchArrayColumn.width`

Width of pitch array column.

Defined equal to 1 when pitch array column contains cells.

Defined equal to 0 when pitch array column contains no cells.

Returns 1 or 0.

Methods

`PitchArrayColumn.append(cell)`

Appends *cell* to pitch array column.

Returns none.

`PitchArrayColumn.extend(cells)`

Extends *cells* against pitch array column.

Returns none.

`PitchArrayColumn.remove_pitches()`

Removes pitches from pitch array cells in pitch array column.

Returns none.

Special methods

`PitchArrayColumn.__eq__(arg)`

True when *arg* is a pitch array column with pitch array cells equal to those of this pitch array column. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`PitchArrayColumn.__getitem__(arg)`

Gets item *arg* from pitch array column.

Returns pitch arrach cell.

`PitchArrayColumn.__ne__(arg)`

True when pitch array column does not equal *arg*. Otherwise false.

Returns boolean.

`PitchArrayColumn.__repr__()`

Gets interpreter representation of pitch array column.

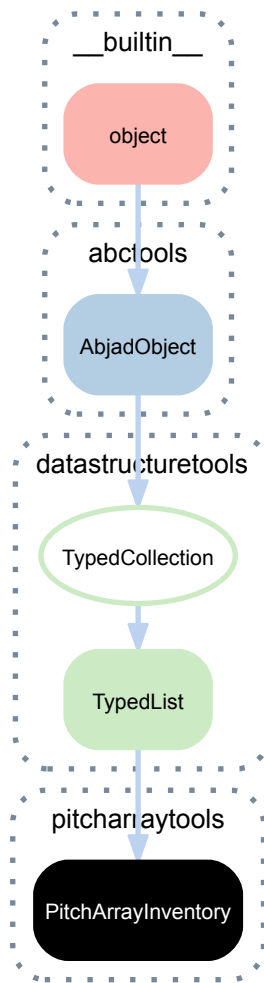
Returns string.

`PitchArrayColumn.__str__()`

String representation of pitch array column.

Returns string.

12.1.4 pitcharraytools.PitchArrayInventory



class `pitcharraytools.PitchArrayInventory` (*tokens=None*, *item_class=None*,
keep_sorted=None, *custom_identifier=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 format(inventory)
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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.

Returns string or none.

`(TypedList).keep_sorted`
Sorts collection on mutation if true.

Methods

`(TypedList).append(token)`
Changes *token* to item and appends.

```
>>> 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)`
Changes *token* to item and returns 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)`
Changes *tokens* to items and extends.

```
>>> 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)`
Changes *token* to item and returns index.

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

Returns index.

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

```
>>> 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.

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

(TypedList) **.remove** (*token*)
Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList).**__delitem__**(*i*)

Aliases list.__delitem__().

(TypedCollection).**__eq__**(*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection).**__format__**(*format_specification*='')

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList).**__getitem__**(*i*)

Aliases list.__getitem__().

(TypedList).**__iadd__**(*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)  
datastructuretools.TypedList(  
    [  
        indicatortools.Dynamic(  
            'ppp'  
        ),  
        indicatortools.Dynamic(  
            'p'  
        ),  
        indicatortools.Dynamic(  
            'mp'  
        ),  
        indicatortools.Dynamic(  
            'mf'  
        ),  
        indicatortools.Dynamic(  
            'fff'  
        ),  
    ],  
    item_class=indicatortools.Dynamic,  
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList).**__reversed__**()

Aliases list.__reversed__().

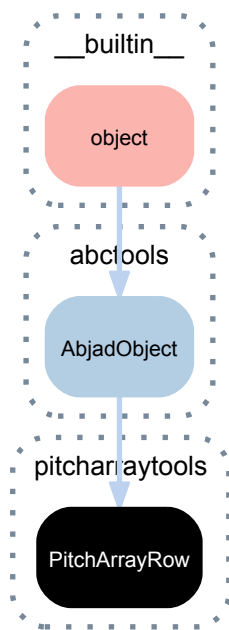
(TypedList).**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```

12.1.5 `pitcharraytools.PitchArrayRow`



class `pitcharraytools.PitchArrayRow`(*cells*)
 A pitch array row.

```
>>> 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'"))
```

Bases

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

Read-only properties

`PitchArrayRow.cell_tokens`
 Cell tokens of pitch array row.

Returns tuple.

`PitchArrayRow.cell_widths`
 Cell widths of pitch array row.

Returns tuple.

`PitchArrayRow.cells`

Cells of pitch array row.

Returns tuple.

`PitchArrayRow.depth`

Depth of pitch array row.

Defined equal to 1.

Returns 1.

`PitchArrayRow.dimensions`

Dimensions of pitch array row.

Returns pair.

`PitchArrayRow.is_defective`

True when width of pitch array row does not equal width of parent pitch array. Otherwise false.

Returns boolean.

`PitchArrayRow.is_in_range`

True when all pitches in pitch array row are in pitch range of pitch array row. Otherwise false.

Returns boolean.

`PitchArrayRow.parent_array`

Parent pitch array housing pitch array row.

Returns pitch array or none.

`PitchArrayRow.pitches`

Pitches in pitch array row.

Returns tuple.

`PitchArrayRow.row_index`

Row index of pitch array row in parent pitch array.

Returns nonnegative integer.

`PitchArrayRow.weight`

Weight of pitch array row.

Defined equal to sum of weights of pitch array cells in pitch array row.

Returns nonnegative integer.

`PitchArrayRow.width`

Width of pitch array row.

Defined equal to sum of widths of pitch array cells in pitch array row.

Returns nonnegative integer.

Read/write properties

`PitchArrayRow.pitch_range`

Gets and set pitch range of pitch array row.

Returns pitch range.

Methods

`PitchArrayRow.append(cell_token)`

Appends *cell_token* to pitch array row.

Returns none.

`PitchArrayRow.apply_pitches (pitch_tokens)`

Applies *pitch_tokens* to pitch cells in pitch array row.

Returns none.

`PitchArrayRow.copy_subrow (start=None, stop=None)`

Copies subrow of pitch array row from *start* to *stop*.

Returns new pitch array row.

`PitchArrayRow.empty_pitches ()`

Empties pitches in pitch array row.

Returns none.

`PitchArrayRow.extend (cell_tokens)`

Extends *cell_tokens* against pitch array row.

Returns none.

`PitchArrayRow.has_spanning_cell_over_index (i)`

True when pitch array row has one or more cells spanning over index *i*. Otherwise false.

Returns boolean.

`PitchArrayRow.index (cell)`

Index of pitch array *cell* in pitch array row.

Returns nonnegative integer.

`PitchArrayRow.merge (cells)`

Merges *cells*.

Returns pitch array cell.

`PitchArrayRow.pad_to_width (width)`

Pads pitch array row to *width*.

Returns none.

`PitchArrayRow.pop (cell_index)`

Pops cell *cell_index* from pitch array row.

Returns pitch array cell.

`PitchArrayRow.remove (cell)`

Removes *cell* from pitch array row.

Returns none.

`PitchArrayRow.to_measure (cell_duration_denominator=8)`

Changes 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 ()`

Withdraws pitch array row from parent pitch array.

Returns pitch array row.

Special methods

`PitchArrayRow.__add__(arg)`
Concatenates *arg* to pitch array row.
Returns new pitch array row.

`PitchArrayRow.__copy__()`
Copies pitch array row.
Returns new pitch array row.

`PitchArrayRow.__eq__(arg)`
True when *arg* is a pitch array row with contents equal to that of this pitch array row. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`PitchArrayRow.__getitem__(arg)`
Gets pitch array cell *arg* from pitch array row.
Returns pitch array cell.

`PitchArrayRow.__iadd__(arg)`
Adds *arg* to pitch array row in place.
Returns pitch array row.

`PitchArrayRow.__len__()`
Length of pitch array row.
Defined equal to the width of pitch array row.
Returns nonnegative integer.

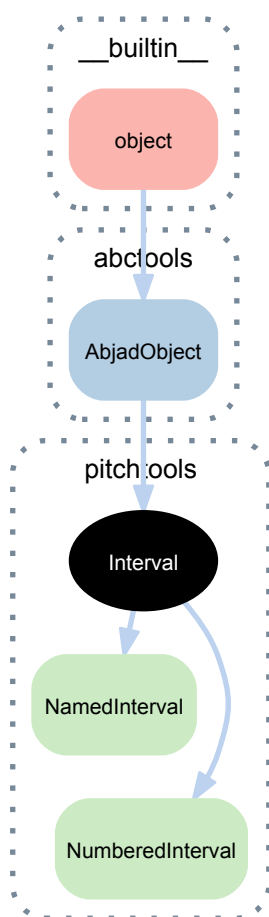
`PitchArrayRow.__ne__(arg)`
True when pitch array row does not equal *arg*. Otherwise false.
Returns boolean.

`PitchArrayRow.__repr__()`
Gets interpreter representation of pitch array row.
Returns string.

`PitchArrayRow.__str__()`
String representation of pitch array row.
Returns string.

13.1 Abstract classes

13.1.1 `pitchtools.Interval`



class `pitchtools.Interval`
Interval base class.

Bases

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

Read-only properties

`Interval.cents`

Cents of interval.

Returns nonnegative number.

`Interval.direction_number`

Direction number of interval.

Returns -1, 0 or 1.

`Interval.direction_string`

Direction string of interval.

Returns string.

`Interval.interval_class`

Interval class of interval.

`Interval.number`

Number of interval.

`Interval.semitones`

Semitones equating to interval.

Return nonnegative number.

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__()`

Absolute value of interval.

Returns new interval.

`Interval.__eq__(arg)`

True when *arg* is an interval with number and direction equal to those of this interval. Otherwise false.

Returns boolean.

`Interval.__float__()`

Change interval to float.

Returns float.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

Interval.**__hash__**()

Hashes interval.

Returns integer.

Interval.**__int__**()

Change interval to integer.

Returns integer.

Interval.**__ne__**(*arg*)

True when interval does not equal *arg*.

Returns boolean.

Interval.**__neg__**()

Negates interval.

Returns interval.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

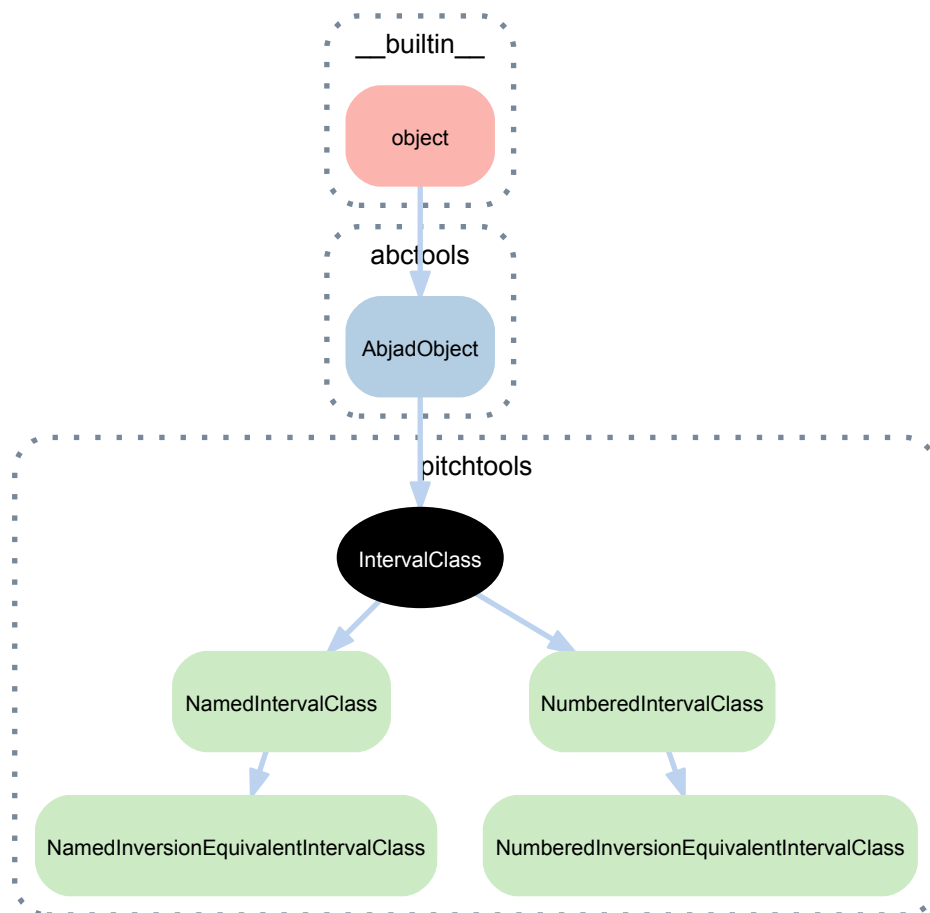
Returns string.

Interval.**__str__**()

String representation of interval.

Returns string.

13.1.2 pitchtools.IntervalClass



class `pitchtools.IntervalClass`
Interval-class base class.

Bases

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

Read-only properties

`IntervalClass.number`
Number of interval-class.
Returns number.

Special methods

`IntervalClass.__abs__()`
Absolute value of interval-class.
Returns new interval-class.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`IntervalClass.__float__()`

Changes interval-class to float.

Returns float.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`IntervalClass.__hash__()`

Hashes interval-class.

Returns integer.

`IntervalClass.__int__()`

Change interval-class to integer.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

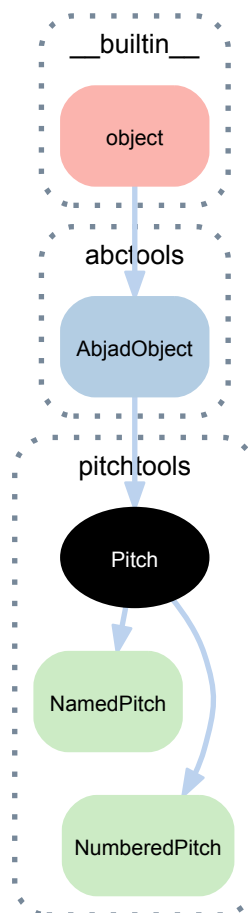
Returns string.

`IntervalClass.__str__()`

String representation of interval-class.

Returns string.

13.1.3 pitchtools.Pitch



class `pitchtools.Pitch`
Pitch base class.

Bases

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

Read-only properties

`Pitch.accidental`
Accidental of pitch.

`Pitch.accidental_spelling`
Accidental spelling of Abjad session.

```
>>> NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

`Pitch.alteration_in_semitones`
Alteration of pitch in semitones.

`Pitch.diatonic_pitch_class_name`
Diatonic pitch-class name of pitch.

`Pitch.diatonic_pitch_class_number`
Diatonic pitch-class number of pitch.

`Pitch.diatonic_pitch_name`
Diatonic pitch name of pitch.

`Pitch.diatonic_pitch_number`
Diatonic pitch number of pitch.

`Pitch.named_pitch`
Named pitch corresponding to pitch.

`Pitch.named_pitch_class`
Named pitch-class corresponding to pitch.

`Pitch.numbered_pitch`
Numbered pitch corresponding to pitch.

`Pitch.numbered_pitch_class`
Numbered pitch-class corresponding to pitch.

`Pitch.octave`
Octave of pitch.
Returns octave.

`Pitch.octave_number`
Octave number of pitch.
Returns integer.

`Pitch.pitch_class_name`
Pitch-class name corresponding to pitch.
Returns string.

`Pitch.pitch_class_number`
Pitch-class number of pitch.
Returns number

`Pitch.pitch_class_octave_label`
Pitch-class / octave label of pitch.

`Pitch.pitch_name`
Pitch name of pitch.
Returns string.

`Pitch.pitch_number`
Pitch number of pitch.
Returns number.

Methods

`Pitch.apply_accidental (accidental=None)`
Applies *accidental* to pitch.
Returns new pitch.

`Pitch.invert (axis=None)`
Inverts pitch about *axis*.
Returns new pitch.

`Pitch.multiply (n=1)`
Multiplies pitch by *n*.

`Pitch.transpose(expr)`
Transposes pitch by *expr*.
Returns new 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
```

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 when *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__()`

Absolute value of pitch.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`Pitch.__float__()`

Changes pitch to float.

Returns float.

`Pitch.__format__(format_specification='')`

Formats pitch.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

`Pitch.__hash__()`

Hashes pitch.

Returns integer.

`Pitch.__illustrate__()`

Illustrates pitch.

Returns LilyPond file.

`Pitch.__int__()`

Changes pitch to integer.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

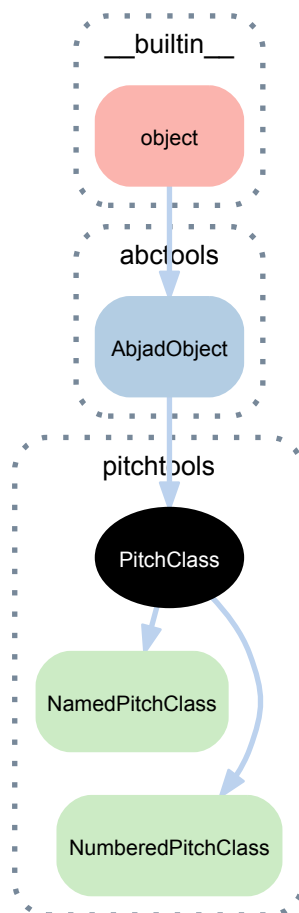
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.1.4 pitchtools.PitchClass



class `pitchtools.PitchClass`
Pitch-class base class.

Bases

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

Read-only properties

`PitchClass.accidental`
Accidental of pitch-class.

`PitchClass.accidental_spelling`
Accidental spelling of pitch-class.
Returns string.

`PitchClass.alteration_in_semitones`
Alteration of pitch-class in semitones.

`PitchClass.diatonic_pitch_class_name`
Diatonic pitch-class name corresponding to pitch-class.

`PitchClass.diatonic_pitch_class_number`
Diatonic pitch-class number corresponding to pitch-class.

`PitchClass.named_pitch_class`

Named pitch-class corresponding to pitch-class.

`PitchClass.numbered_pitch_class`

Numbered pitch-class corresponding to pitch-class.

`PitchClass.pitch_class_label`

Pitch-class label of pitch-class.

`PitchClass.pitch_class_name`

Pitch-class name of pitch-class.

`PitchClass.pitch_class_number`

Pitch-class number of pitch-class.

Methods

`PitchClass.apply_accidental (accidental=None)`

Applies *accidental* to pitch-class.

Returns new pitch-class.

`PitchClass.invert (axis=None)`

Inverts pitch-class about *axis*.

Returns new pitch-class.

`PitchClass.multiply (n=1)`

Multiplies pitch-class by *n*.

Returns new pitch-class.

`PitchClass.transpose (expr)`

Transposes pitch-class by *n*'.

Returns new 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

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

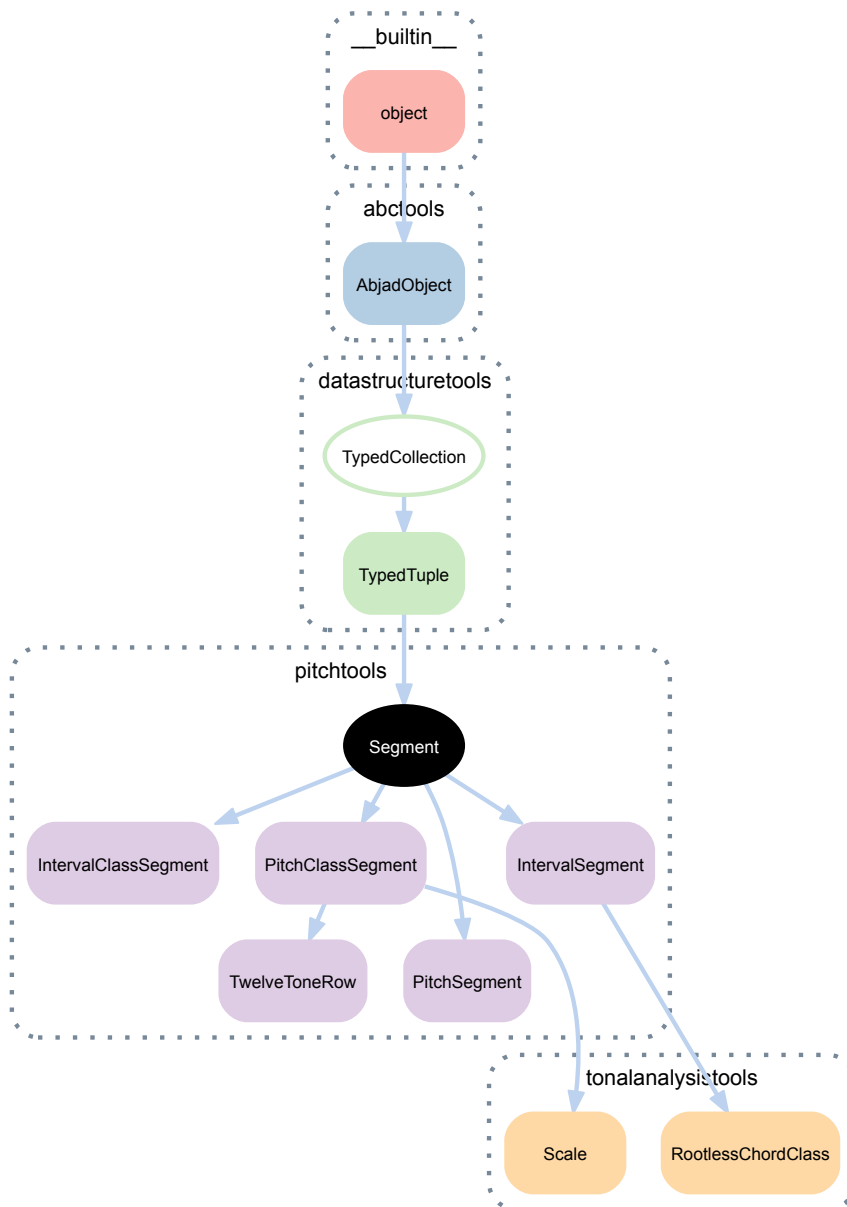
`PitchClass.__format__(format_specification='')`
Formats component.
Set *format_specification* to `'`, `'lilypond'` or `'storage'`.
Returns string.

`PitchClass.__hash__()`
Has pitch-class.
Returns integer.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

13.1.5 pitchtools.Segment



class `pitchtools.Segment` (*tokens=None, item_class=None, custom_identifier=None*)
 Music-theoretic segment base class.

Bases

- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`Segment.has_duplicates`
 True when segment has duplicates. Otherwise false.

Returns boolean.

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

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

`(TypedTuple).count(token)`
Changes *token* to item.

Returns count in collection.

`Segment.from_selection(selection, item_class=None, custom_identifier=None)`
Makes segment from *selection*.

Returns new segment.

`(TypedTuple).index(token)`
Changes *token* to item.

Returns index in collection.

Special methods

`(TypedTuple).__add__(expr)`
Adds typed tuple to *expr*.

Returns new typed tuple.

`(TypedTuple).__contains__(token)`
Change *token* to item and return true if item exists in collection.

Returns none.

`(TypedCollection).__eq__(expr)`
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`
Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TypedTuple).__getitem__(i)`
Gets *i* from type tuple.

Returns item.

`(TypedTuple).__getslice__(start, stop)`
Gets slice from *start* to *stop* in typed tuple.

Returns new typed tuple.

(TypedTuple) .**__hash__**()

Hashes typed tuple.

Returns integer.

(TypedCollection) .**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedCollection) .**__makenew__** (*tokens=None, item_class=None, custom_identifier=None*)

Makes new typed collection with optional new values.

Returns new typed collection.

(TypedTuple) .**__mul__** (*expr*)

Multiplies typed tuple by *expr*.

Returns new typed tuple.

(TypedCollection) .**__ne__** (*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedTuple) .**__rmul__** (*expr*)

Multiplies *expr* by typed tuple.

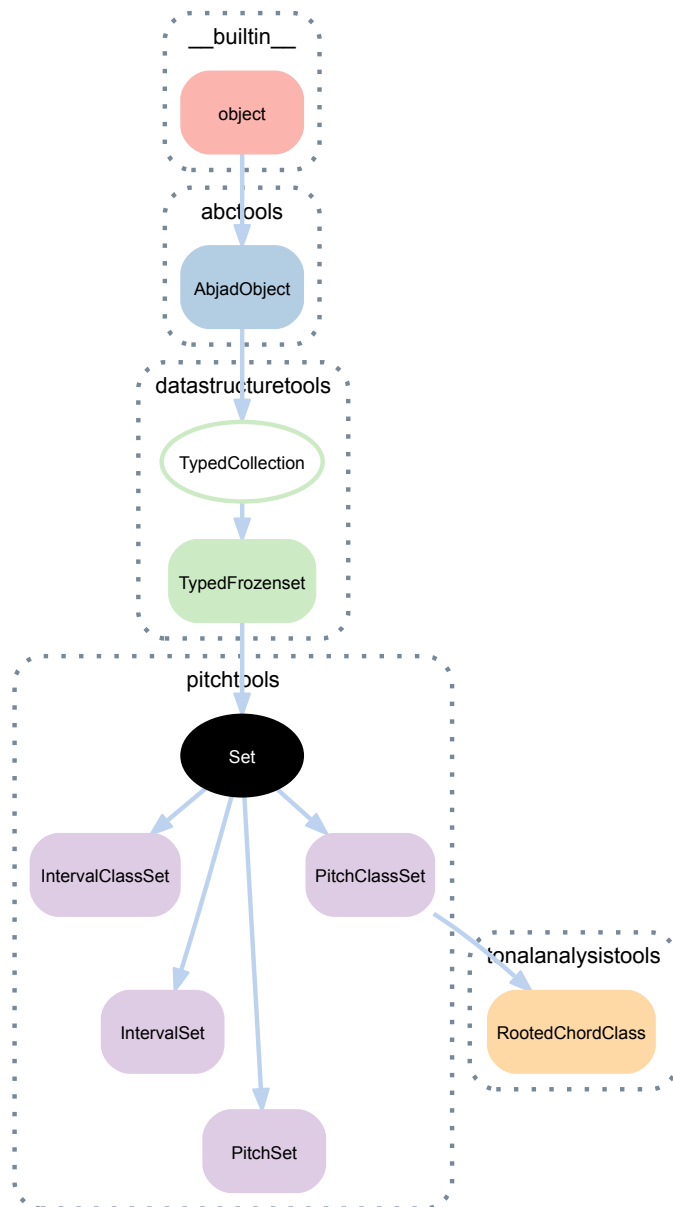
Returns new typed tuple.

Segment .**__str__**()

String representation of segment.

Returns string.

13.1.6 pitchtools.Set



class `pitchtools.Set` (*tokens=None, item_class=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
 Gets and sets custom identifier of typed collection.
 Returns string or none.

Methods

`(TypedFrozenSet).copy()`
 Copies typed frozen set.

Returns new typed frozen set.

`(TypedFrozenSet).difference(expr)`
 Typed frozen set set-minus *expr*.

Returns new typed frozen set.

`Set.from_selection(selection, item_class=None, custom_identifier=None)`
 Makes set from *selection*.

Returns set.

`(TypedFrozenSet).intersection(expr)`
 Set-theoretic intersection of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedFrozenSet).isdisjoint(expr)`
 True when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).issubset(expr)`
 True when typed frozen set is a subset of *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).issuperset(expr)`
 True when typed frozen set is a superset of *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).symmetric_difference(expr)`
 Symmetric difference of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedFrozenSet).union(expr)`
 Union of typed frozen set and *expr*.

Returns new typed frozen set.

Special methods

`(TypedFrozenSet).__and__(expr)`
 Logical AND of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedCollection).__contains__(token)`
 True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedCollection) .**__eq__**(*expr*)
 True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.
 Returns boolean.

(TypedCollection) .**__format__**(*format_specification*='')
 Formats typed collection.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(TypedFrozenSet) .**__ge__**(*expr*)
 True when typed frozen set is greater than or equal to *expr*. Otherwise false.
 Returns boolean.

(TypedFrozenSet) .**__gt__**(*expr*)
 True when typed frozen set is greater than *expr*. Otherwise false.
 Returns boolean.

(TypedFrozenSet) .**__hash__**()
 Hashes typed frozen set.
 Returns integer.

(TypedCollection) .**__iter__**()
 Iterates typed collection.
 Returns generator.

(TypedFrozenSet) .**__le__**(*expr*)
 True when typed frozen set is less than or equal to *expr*. Otherwise false.
 Returns boolean.

(TypedCollection) .**__len__**()
 Length of typed collection.
 Returns nonnegative integer.

(TypedFrozenSet) .**__lt__**(*expr*)
 True when typed frozen set is less than *expr*. Otherwise false.
 Returns boolean.

(TypedCollection) .**__makenew__**(*tokens=None, item_class=None, custom_identifier=None*)
 Makes new typed collection with optional new values.
 Returns new typed collection.

(TypedFrozenSet) .**__ne__**(*expr*)
 True when typed frozen set is not equal to *expr*. Otherwise false.
 Returns boolean.

(TypedFrozenSet) .**__or__**(*expr*)
 Logical OR of typed frozen set and *expr*.
 Returns new typed frozen set.

(AbjadObject) .**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

Set .**__str__**()
 String representation of set.
 Returns string.

`(TypedFrozenSet) .__sub__(expr)`

Subtracts *expr* from typed frozen set.

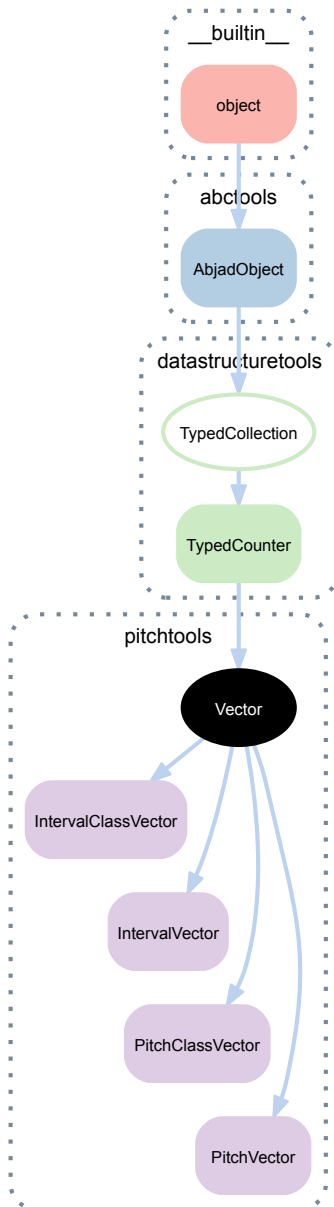
Returns new typed frozen set.

`(TypedFrozenSet) .__xor__(expr)`

Logical XOR of typed frozen set and *expr*.

Returns new typed frozen set.

13.1.7 pitchtools.Vector



class `pitchtools.Vector` (*tokens=None, item_class=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

`(TypedCounter).clear()`
Clears typed counter.

Returns none.

`(TypedCounter).copy()`
Copies typed counter.

Returns new typed counter.

`(TypedCounter).elements()`
Elements in typed counter.

`Vector.from_selection(selection, item_class=None, custom_identifier=None)`
Makes vector from *selection*.

Returns vector.

`(TypedCounter).items()`
Items in typed counter.

Returns tuple.

`(TypedCounter).iteritems()`
Iterates items in typed counter.

Yields items.

`(TypedCounter).iterkeys()`
Iterates keys in typed counter.

`(TypedCounter).itervalues()`
Iterates values in typed counter.

`(TypedCounter).keys()`
Keys in typed counter.

`(TypedCounter).most_common(n=None)`
Please document.

`(TypedCounter).subtract(iterable=None, **kwargs)`
Stracts *iterable* from typed counter.

`(TypedCounter).update(iterable=None, **kwargs)`
Updates typed counter with *iterable*.

`(TypedCounter).values()`
Values of typed counter.

(TypedCounter).**viewitems**()
Please document.

(TypedCounter).**viewkeys**()
Please document.

(TypedCounter).**viewvalues**()
Please document.

Special methods

(TypedCounter).**__add__**(*expr*)
Adds typed counter to *expr*.

Returns new typed counter.

(TypedCounter).**__and__**(*expr*)
Logical AND of typed counter and *expr*.

Returns new typed counter.

(TypedCollection).**__contains__**(*token*)
True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedCounter).**__delitem__**(*token*)
Deletes *token* from typed counter.

Returns none.

(TypedCollection).**__eq__**(*expr*)
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection).**__format__**(*format_specification*='')
Formats typed collection.

Set *format_specification* to ' or 'storage'. Interprets ' equal to 'storage'.

Returns string.

(TypedCounter).**__getitem__**(*token*)
Gets *token* from typed counter.

Returns item.

(TypedCollection).**__iter__**()
Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()
Length of typed collection.

Returns nonnegative integer.

(TypedCollection).**__makenew__**(*tokens=None, item_class=None, custom_identifier=None*)
Makes new typed collection with optional new values.

Returns new typed collection.

(TypedCounter).**__missing__**(*token*)
Returns zero.

Returns zero.

(TypedCollection).**__ne__**(*expr*)
 True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.
 Returns boolean.

(TypedCounter).**__or__**(*expr*)
 Logical OR of typed counter and *expr*.
 Returns new typed counter.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

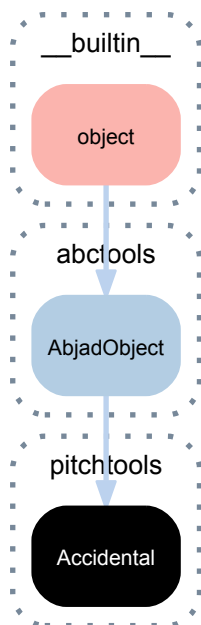
(TypedCounter).**__setitem__**(*token*, *value*)
 Sets typed counter *token* to *value*.
 Returns none.

Vector.**__str__**()
 String representation of vector.
 Returns string.

(TypedCounter).**__sub__**(*expr*)
 Subtracts *expr* from typed counter.
 Returns new typed counter.

13.2 Concrete classes

13.2.1 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.abbreviation`

Abbreviation of accidental.

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.abbreviation
's'
```

Returns string.

`Accidental.is_adjusted`

True for all accidentals equal to a nonzero number of semitones. Otherwise false:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.is_adjusted
True
```

Returns boolean.

`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_string`

Symbolic string of accidental.

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.symbolic_string
'#'
```

Returns string.

Static methods

`Accidental.is_abbreviation(expr)`

True when *expr* is an alphabetic accidental abbreviation. Otherwise false:

```
>>> pitchtools.Accidental.is_abbreviation('tqs')
True
```

The regex `^([s]{1,2}|[f]{1,2}|t?q?[fs])!?$` underlies this predicate.

Returns boolean.

`Accidental.is_symbolic_string(expr)`

True when *expr* is a symbolic accidental string. Otherwise false:

```
>>> pitchtools.Accidental.is_symbolic_string('#+')
True
```

True on empty string.

The regex `^([\#]{1,2}|[b]{1,2}|[\#]?[+]|[b]?[~]|)$` underlies this predicate.

Returns boolean.

Special methods

`Accidental.__add__(arg)`

Adds *arg* to accidental.

Returns new accidental.

`Accidental.__eq__(arg)`

True when *arg* is an accidental with an abbreviation equal to that of this accidental. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`Accidental.__ge__(arg)`

True when *arg* is an accidental with semitones less than or equal to those of this accidental. Otherwise false.

Returns boolean.

`Accidental.__gt__(arg)`

True when *arg* is an accidental with semitones less than those of this accidental. Otherwise false.

Returns boolean.

`Accidental.__le__(arg)`

True when *arg* is an accidental with semitones greater than or equal to those of this accidental. Otherwise false.

Returns boolean.

`Accidental.__lt__(arg)`

True when *arg* is an accidental with semitones greater than those of this accidental. Otherwise false.

Returns boolean.

`Accidental.__ne__(arg)`

True when accidental does not equal *arg*. Otherwise false.

Returns boolean.

`Accidental.__neg__()`

Negates accidental.

Returns new accidental.

`Accidental.__nonzero__()`

Defined equal to true.

Returns true.

`(AbjadObject).__repr__()`

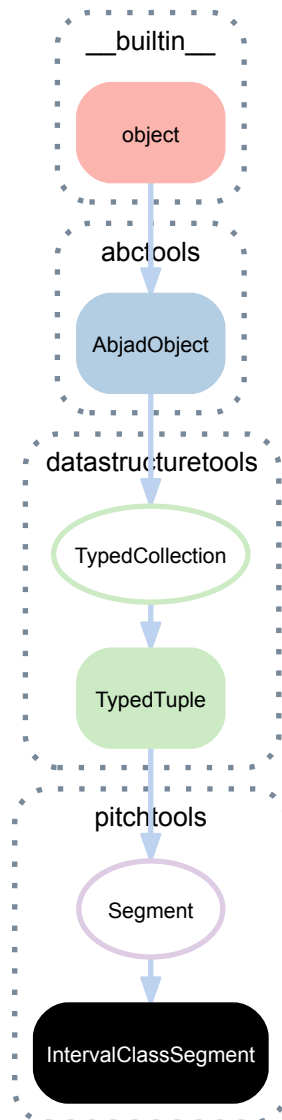
Gets interpreter representation of Abjad object.

Returns string.

`Accidental.__str__()`
 String representation of accidental.
 Returns string.

`Accidental.__sub__(arg)`
 Subtracts *arg* from accidental.
 Returns new accidental.

13.2.2 `pitchtools.IntervalClassSegment`



`class pitchtools.IntervalClassSegment` (*tokens=None, item_class=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`

Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

`(TypedTuple).count(token)`

Changes *token* to item.

Returns count in collection.

`(TypedTuple).index(token)`

Changes *token* to item.

Returns index in collection.

Class methods

`IntervalClassSegment.from_selection(selection, item_class=None, cus-
tom_identifier=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)`

Adds typed tuple to *expr*.

Returns new typed tuple.

`(TypedTuple).__contains__(token)`

Change *token* to item and return true if item exists in collection.

Returns none.

`(TypedCollection).__eq__(expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`

Formats typed collection.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(TypedTuple).__getitem__(i)`

Gets *i* from type tuple.

Returns item.

`(TypedTuple).__getslice__(start, stop)`

Gets slice from *start* to *stop* in typed tuple.

Returns new typed tuple.

`(TypedTuple).__hash__()`

Hashes typed tuple.

Returns integer.

`(TypedCollection).__iter__()`

Iterates typed collection.

Returns generator.

`(TypedCollection).__len__()`

Length of typed collection.

Returns nonnegative integer.

`(TypedCollection).__makenew__(tokens=None, item_class=None, custom_identifier=None)`

Makes new typed collection with optional new values.

Returns new typed collection.

(TypedTuple) .**__mul__**(*expr*)

Multiplies typed tuple by *expr*.

Returns new typed tuple.

(TypedCollection) .**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedTuple) .**__rmul__**(*expr*)

Multiplies *expr* by typed tuple.

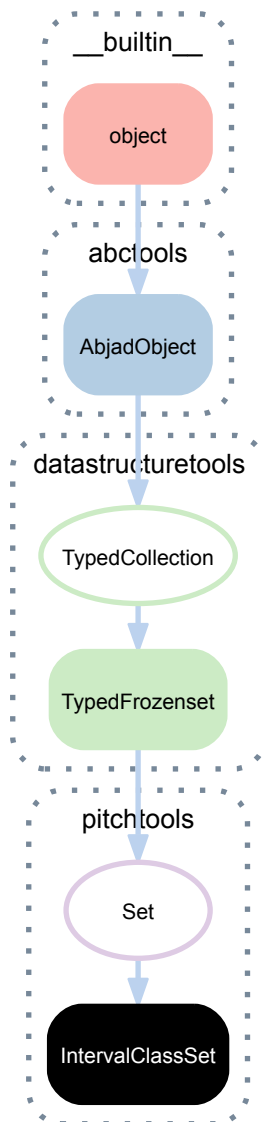
Returns new typed tuple.

(Segment) .**__str__**()

String representation of segment.

Returns string.

13.2.3 pitchtools.IntervalClassSet



class `pitchtools.IntervalClassSet` (*tokens=None, item_class=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

Methods

`(TypedFrozenSet).copy()`
Copies typed frozen set.
Returns new typed frozen set.

`(TypedFrozenSet).difference(expr)`
Typed frozen set set-minus *expr*.
Returns new typed frozen set.

`(TypedFrozenSet).intersection(expr)`
Set-theoretic intersection of typed frozen set and *expr*.
Returns new typed frozen set.

`(TypedFrozenSet).isdisjoint(expr)`
True when typed frozen set shares no elements with *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).issubset(expr)`
True when typed frozen set is a subset of *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).issuperset(expr)`
True when typed frozen set is a superset of *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).symmetric_difference(expr)`
Symmetric difference of typed frozen set and *expr*.
Returns new typed frozen set.

`(TypedFrozenSet).union(expr)`
Union of typed frozen set and *expr*.
Returns new typed frozen set.

Class methods

`IntervalClassSet.from_selection(selection, item_class=None, custom_identifier=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)`

Logical AND of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedCollection) .__contains__(token)`

True when typed collection container *token*. Otherwise false.

Returns boolean.

`(TypedCollection) .__eq__(expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection) .__format__(format_specification='')`

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TypedFrozenSet) .__ge__(expr)`

True when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet) .__gt__(expr)`

True when typed frozen set is greater than *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet) .__hash__()`

Hashes typed frozen set.

Returns integer.

`(TypedCollection) .__iter__()`

Iterates typed collection.

Returns generator.

`(TypedFrozenSet) .__le__(expr)`

True when typed frozen set is less than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedCollection) .__len__()`

Length of typed collection.

Returns nonnegative integer.

(TypedFrozenSet) .**__lt__**(*expr*)
True when typed frozen set is less than *expr*. Otherwise false.
Returns boolean.

(TypedCollection) .**__makenew__**(*tokens=None, item_class=None, custom_identifier=None*)
Makes new typed collection with optional new values.
Returns new typed collection.

(TypedFrozenSet) .**__ne__**(*expr*)
True when typed frozen set is not equal to *expr*. Otherwise false.
Returns boolean.

(TypedFrozenSet) .**__or__**(*expr*)
Logical OR of typed frozen set and *expr*.
Returns new typed frozen set.

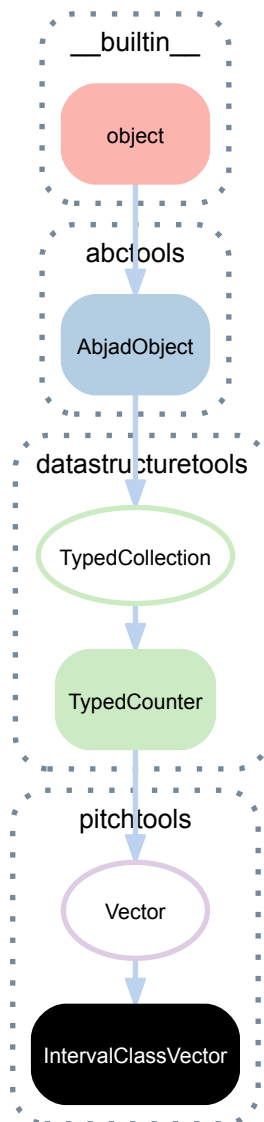
(AbjadObject) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(Set) .**__str__**()
String representation of set.
Returns string.

(TypedFrozenSet) .**__sub__**(*expr*)
Subtracts *expr* from typed frozen set.
Returns new typed frozen set.

(TypedFrozenSet) .**__xor__**(*expr*)
Logical XOR of typed frozen set and *expr*.
Returns new typed frozen set.

13.2.4 pitchtools.IntervalClassVector



class `pitchtools.IntervalClassVector` (*tokens=None*, *item_class=None*, *custom_identifier=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
  
```

Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

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

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

`(TypedCounter).clear()`
Clears typed counter.

Returns none.

`(TypedCounter).copy()`
Copies typed counter.

Returns new typed counter.

`(TypedCounter).elements()`
Elements in typed counter.

`(TypedCounter).items()`
Items in typed counter.

Returns tuple.

`(TypedCounter).iteritems()`
Iterates items in typed counter.

Yields items.

`(TypedCounter).iterkeys()`
Iterates keys in typed counter.

`(TypedCounter).intervalues()`
Iterates values in typed counter.

`(TypedCounter).keys()`
Keys in typed counter.

`(TypedCounter).most_common(n=None)`
Please document.

`(TypedCounter).subtract(iterable=None, **kwargs)`
Stracts *iterable* from typed counter.

`(TypedCounter).update(iterable=None, **kwargs)`

Updates typed counter with *iterable*.

`(TypedCounter).values()`

Values of typed counter.

`(TypedCounter).viewitems()`

Please document.

`(TypedCounter).viewkeys()`

Please document.

`(TypedCounter).viewvalues()`

Please document.

Class methods

`IntervalClassVector.from_selection(selection, item_class=None, custom_identifier=None)`

Makes interval-class vector from *selection*.

Returns interval-class vector.

Special methods

`(TypedCounter).__add__(expr)`

Adds typed counter to *expr*.

Returns new typed counter.

`(TypedCounter).__and__(expr)`

Logical AND of typed counter and *expr*.

Returns new typed counter.

`(TypedCollection).__contains__(token)`

True when typed collection container *token*. Otherwise false.

Returns boolean.

`(TypedCounter).__delitem__(token)`

Deletes *token* from typed counter.

Returns none.

`(TypedCollection).__eq__(expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TypedCounter).__getitem__(token)`

Gets *token* from typed counter.

Returns item.

`(TypedCollection).__iter__()`

Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedCollection) .**__makenew__** (*tokens=None, item_class=None, custom_identifier=None*)

Makes new typed collection with optional new values.

Returns new typed collection.

(TypedCounter) .**__missing__** (*token*)

Returns zero.

Returns zero.

(TypedCollection) .**__ne__** (*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(TypedCounter) .**__or__** (*expr*)

Logical OR of typed counter and *expr*.

Returns new typed counter.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedCounter) .**__setitem__** (*token, value*)

Sets typed counter *token* to *value*.

Returns none.

(Vector) .**__str__**()

String representation of vector.

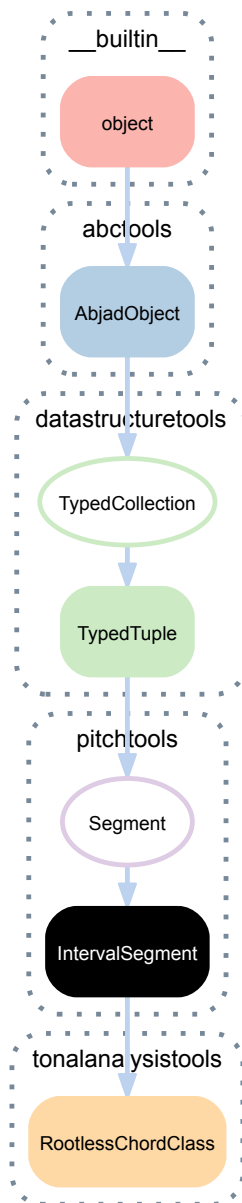
Returns string.

(TypedCounter) .**__sub__** (*expr*)

Subtracts *expr* from typed counter.

Returns new typed counter.

13.2.5 pitchtools.IntervalSegment



class `pitchtools.IntervalSegment` (*tokens=None, item_class=None, custom_identifier=None*)
 An interval segment.

```
>>> intervals = 'm2 M10 -aug4 P5'
>>> pitchtools.IntervalSegment(intervals)
IntervalSegment(['+m2', '+M10', '-aug4', '+P5'])
```

Bases

- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`IntervalSegment.has_duplicates`

True if segment has duplicate items. Otherwise false.

```
>>> 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`

Slope of interval segment.

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`

Spread of interval segment.

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.

Read/write properties

`(TypedCollection).custom_identifier`

Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

`(TypedTuple).count(token)`

Changes *token* to item.

Returns count in collection.

`(TypedTuple).index(token)`

Changes *token* to item.

Returns index in collection.

`IntervalSegment.rotate(n)`

Rotates interval segment by *n*.

Returns new interval segment.

Class methods

`IntervalSegment.from_selection(selection, item_class=None, custom_identifier=None)`
 Makes 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)`
 Adds typed tuple to *expr*.

Returns new typed tuple.

`(TypedTuple).__contains__(token)`
 Change *token* to item and return true if item exists in collection.

Returns none.

`(TypedCollection).__eq__(expr)`
 True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`
 Formats typed collection.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TypedTuple).__getitem__(i)`
 Gets *i* from type tuple.

Returns item.

`(TypedTuple).__getslice__(start, stop)`
 Gets slice from *start* to *stop* in typed tuple.

Returns new typed tuple.

`(TypedTuple).__hash__()`
 Hashes typed tuple.

Returns integer.

`(TypedCollection).__iter__()`
 Iterates typed collection.

Returns generator.

`(TypedCollection).__len__()`
 Length of typed collection.

Returns nonnegative integer.

`(TypedCollection).__makenew__(tokens=None, item_class=None, custom_identifier=None)`
 Makes new typed collection with optional new values.

Returns new typed collection.

(TypedTuple) .**__mul__**(*expr*)

Multiplies typed tuple by *expr*.

Returns new typed tuple.

(TypedCollection) .**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedTuple) .**__rmul__**(*expr*)

Multiplies *expr* by typed tuple.

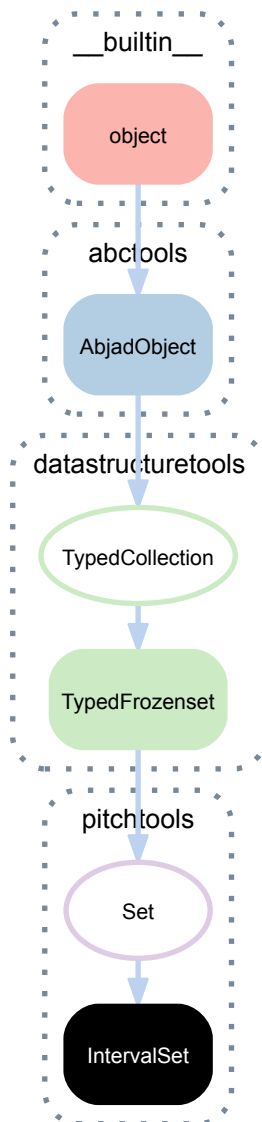
Returns new typed tuple.

(Segment) .**__str__**()

String representation of segment.

Returns string.

13.2.6 pitchtools.IntervalSet



class `pitchtools.IntervalSet` (*tokens=None, item_class=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

Methods

`(TypedFrozenSet).copy()`
Copies typed frozen set.
Returns new typed frozen set.

`(TypedFrozenSet).difference(expr)`
Typed frozen set set-minus *expr*.
Returns new typed frozen set.

`(TypedFrozenSet).intersection(expr)`
Set-theoretic intersection of typed frozen set and *expr*.
Returns new typed frozen set.

`(TypedFrozenSet).isdisjoint(expr)`
True when typed frozen set shares no elements with *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).issubset(expr)`
True when typed frozen set is a subset of *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).issuperset(expr)`
True when typed frozen set is a superset of *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).symmetric_difference(expr)`
Symmetric difference of typed frozen set and *expr*.
Returns new typed frozen set.

`(TypedFrozenSet).union(expr)`
Union of typed frozen set and *expr*.
Returns new typed frozen set.

Class methods

`IntervalSet.from_selection(selection, item_class=None, custom_identifier=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)`

Logical AND of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedCollection) .__contains__ (token)`

True when typed collection container *token*. Otherwise false.

Returns boolean.

`(TypedCollection) .__eq__ (expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection) .__format__ (format_specification='')`

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TypedFrozenSet) .__ge__ (expr)`

True when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet) .__gt__ (expr)`

True when typed frozen set is greater than *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet) .__hash__ ()`

Hashes typed frozen set.

Returns integer.

`(TypedCollection) .__iter__ ()`

Iterates typed collection.

Returns generator.

`(TypedFrozenSet) .__le__ (expr)`

True when typed frozen set is less than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedCollection) .__len__ ()`

Length of typed collection.

Returns nonnegative integer.

(TypedFrozenSet) .**__lt__**(*expr*)
True when typed frozen set is less than *expr*. Otherwise false.
Returns boolean.

(TypedCollection) .**__makenew__**(*tokens=None, item_class=None, custom_identifier=None*)
Makes new typed collection with optional new values.
Returns new typed collection.

(TypedFrozenSet) .**__ne__**(*expr*)
True when typed frozen set is not equal to *expr*. Otherwise false.
Returns boolean.

(TypedFrozenSet) .**__or__**(*expr*)
Logical OR of typed frozen set and *expr*.
Returns new typed frozen set.

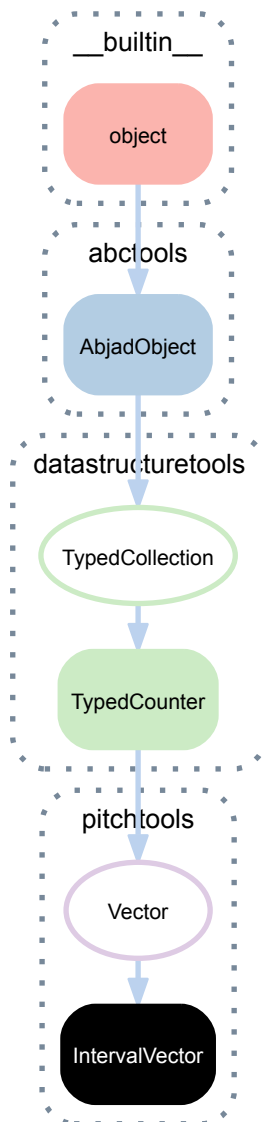
(AbjadObject) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(Set) .**__str__**()
String representation of set.
Returns string.

(TypedFrozenSet) .**__sub__**(*expr*)
Subtracts *expr* from typed frozen set.
Returns new typed frozen set.

(TypedFrozenSet) .**__xor__**(*expr*)
Logical XOR of typed frozen set and *expr*.
Returns new typed frozen set.

13.2.7 pitchtools.IntervalVector



class `pitchtools.IntervalVector` (*tokens=None, item_class=None, custom_identifier=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
```

Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

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

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

Methods

`(TypedCounter).clear()`
Clears typed counter.
Returns none.

`(TypedCounter).copy()`
Copies typed counter.
Returns new typed counter.

`(TypedCounter).elements()`
Elements in typed counter.

`(TypedCounter).items()`
Items in typed counter.
Returns tuple.

`(TypedCounter).iteritems()`
Iterates items in typed counter.
Yields items.

`(TypedCounter).iterkeys()`
Iterates keys in typed counter.

(TypedCounter).**intervalvalues**()
Iterates values in typed counter.

(TypedCounter).**keys**()
Keys in typed counter.

(TypedCounter).**most_common**(*n=None*)
Please document.

(TypedCounter).**subtract**(*iterable=None, **kwargs*)
Stracts *iterable* from typed counter.

(TypedCounter).**update**(*iterable=None, **kwargs*)
Updates typed counter with *iterable*.

(TypedCounter).**values**()
Values of typed counter.

(TypedCounter).**viewitems**()
Please document.

(TypedCounter).**viewkeys**()
Please document.

(TypedCounter).**viewvalues**()
Please document.

Class methods

IntervalVector.**from_selection**(*selection, item_class=None, custom_identifier=None*)
Makes interval vector from *selection*.

Returns interval vector.

Special methods

(TypedCounter).**__add__**(*expr*)
Adds typed counter to *expr*.

Returns new typed counter.

(TypedCounter).**__and__**(*expr*)
Logical AND of typed counter and *expr*.

Returns new typed counter.

(TypedCollection).**__contains__**(*token*)
True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedCounter).**__delitem__**(*token*)
Deletes *token* from typed counter.

Returns none.

(TypedCollection).**__eq__**(*expr*)
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection).**__format__**(*format_specification=''*)
Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedCounter) .**__getitem__** (*token*)
Gets *token* from typed counter.

Returns item.

(TypedCollection) .**__iter__** ()
Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__** ()
Length of typed collection.

Returns nonnegative integer.

(TypedCollection) .**__makenew__** (*tokens=None, item_class=None, custom_identifier=None*)
Makes new typed collection with optional new values.

Returns new typed collection.

(TypedCounter) .**__missing__** (*token*)
Returns zero.

Returns zero.

(TypedCollection) .**__ne__** (*expr*)
True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.
Returns boolean.

(TypedCounter) .**__or__** (*expr*)
Logical OR of typed counter and *expr*.
Returns new typed counter.

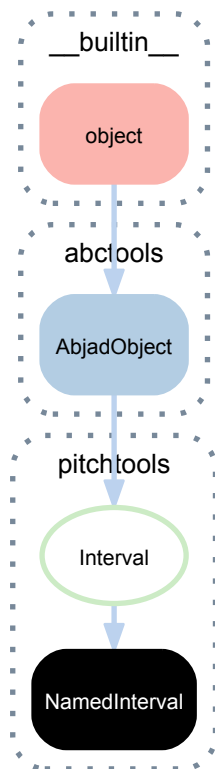
(AbjadObject) .**__repr__** ()
Gets interpreter representation of Abjad object.
Returns string.

(TypedCounter) .**__setitem__** (*token, value*)
Sets typed counter *token* to *value*.
Returns none.

(Vector) .**__str__** ()
String representation of vector.
Returns string.

(TypedCounter) .**__sub__** (*expr*)
Subtracts *expr* from typed counter.
Returns new typed counter.

13.2.8 `pitchtools.NamedInterval`



class `pitchtools.NamedInterval` (*args)
 A named interval.

```
>>> interval = pitchtools.NamedInterval('+M9')
>>> interval
NamedInterval('+M9')
```

Bases

- `pitchtools.Interval`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(Interval).cents`
 Cents of interval.

Returns nonnegative number.

`NamedInterval.direction_number`
 Direction number of named interval.

```
>>> interval.direction_number
1
```

Returns -1, 0 or 1.

`NamedInterval.direction_string`
 Direction string of named interval.

```
>>> interval.direction_string
'ascending'
```

Returns 'ascending', 'descending' or none.

NamedInterval.interval_class
Interval class of named interval.

```
>>> interval.interval_class
2
```

Returns nonnegative integer.

NamedInterval.interval_string
Interval string of named interval.

```
>>> interval.interval_string
'ninth'
```

Returns string.

NamedInterval.named_interval_class
Named interval class of named interval.

```
>>> interval.named_interval_class
NamedInversionEquivalentIntervalClass('+M2')
```

Returns named inversion-equivalent interval-class.

NamedInterval.number
Number of named interval.

```
>>> interval.number
9
```

Returns nonnegative number.

NamedInterval.quality_string
Quality string of named interval.

```
>>> interval.quality_string
'major'
```

Returns string.

NamedInterval.semitones
Semitones of named interval.

```
>>> interval.semitones
14
```

Returns number.

NamedInterval.staff_spaces
Staff spaces of named interval.

```
>>> interval.staff_spaces
8
```

Returns nonnegative integer.

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(
...     NamedPitch(-2),
...     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__()`
 Absolute value of named interval.

```
>>> abs(interval)
NamedInterval('+M9')
```

Returns named interval.

`NamedInterval.__add__(arg)`
 Adds *arg* to named interval.

```
>>> interval + pitchtools.NamedInterval('M2')
NamedInterval('+M10')
```

Returns new named interval.

`NamedInterval.__copy__(*args)`
 Copies named interval.

```
>>> import copy
>>> copy.copy(interval)
NamedInterval('+M9')
```

Returns new named interval.

`NamedInterval.__eq__(arg)`
 True when *arg* is a named interval with a quality string and number equal to this named interval.

```
>>> interval == pitchtools.NamedInterval('+M9')
True
```

Otherwise false:

```
>>> interval == pitchtools.NamedInterval('-M9')
False
```

Returns boolean.

`NamedInterval.__float__()`
Changes number of named interval to a float.

```
>>> float(interval)
9.0
```

Returns float.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`NamedInterval.__ge__(other)`
`x.__ge__(y) <==> x>=y`

`NamedInterval.__gt__(other)`
`x.__gt__(y) <==> x>y`

`(Interval).__hash__()`
Hashes interval.

Returns integer.

`NamedInterval.__int__()`
Returns number of named interval.

```
>>> int(interval)
9
```

Returns integer.

`NamedInterval.__le__(other)`
`x.__le__(y) <==> x<=y`

`NamedInterval.__lt__(arg)`
True when *arg* is a named interval with a number greater than that of this named interval.

```
>>> interval < pitchtools.NamedInterval('+M10')
True
```

Also true when *arg* is a named interval with a number equal to this named interval and with semitones greater than this named interval:

```
>>> pitchtools.NamedInterval('+m9') < interval
True
```

Otherwise false:

```
>>> interval < pitchtools.NamedInterval('+M2')
False
```

Returns boolean.

`NamedInterval.__mul__(arg)`
Multiplies named interval by *arg*.

```
>>> 3 * interval
NamedInterval('+aug25')
```

Returns new named interval.

`NamedInterval.__ne__(arg)`

True when *arg* does not equal this named interval. Otherwise false.

Returns boolean.

`NamedInterval.__neg__()`

Negates named interval.

```
>>> -interval
NamedInterval('-M9')
```

Returns new named interval.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

`NamedInterval.__rmul__(arg)`

Multiplies *arg* by named interval.

```
>>> interval * 3
NamedInterval('+aug25')
```

Returns new named interval.

`NamedInterval.__str__()`

String representation of named interval.

```
>>> str(interval)
'+M9'
```

Returns string.

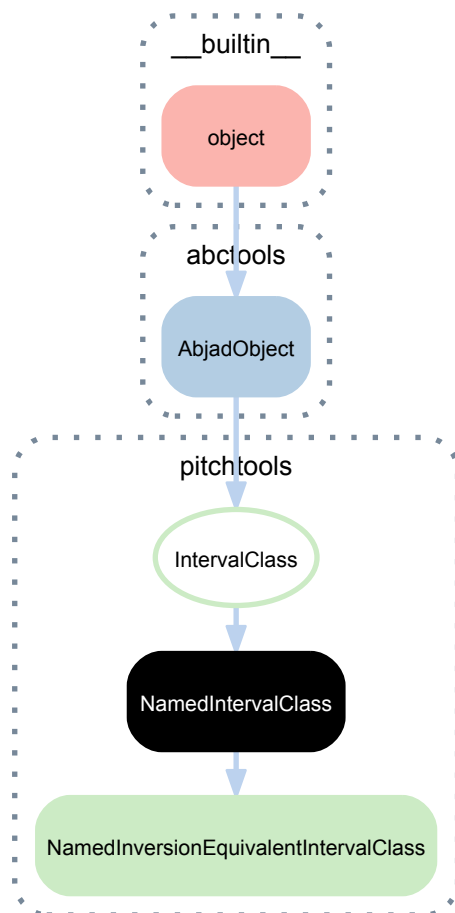
`NamedInterval.__sub__(arg)`

Subtracts *arg* from named interval.

```
>>> interval - pitchtools.NamedInterval('+M2')
NamedInterval('+P8')
```

Returns new named interval.

13.2.9 `pitchtools.NamedIntervalClass`



class `pitchtools.NamedIntervalClass(*args)`
 A named interval-class.

```
>>> pitchtools.NamedIntervalClass('-M9')
NamedIntervalClass('-M2')
```

Bases

- `pitchtools.IntervalClass`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`NamedIntervalClass.direction_number`
 Direction number of named interval-class.

Returns -1, 0 or 1.

`NamedIntervalClass.direction_symbol`
 Direction symbol of named interval-class.

Returns string.

`NamedIntervalClass.direction_word`
 Direction word of named interval-class.

Returns string.

`(IntervalClass).number`
Number of interval-class.

Returns number.

`NamedIntervalClass.quality_string`
Quality string of named interval-class.

Returns string.

Class methods

`NamedIntervalClass.from_pitch_carriers` (*pitch_carrier_1*, *pitch_carrier_2*)
Makes named interval-class from *pitch_carrier_1* and *pitch_carrier_2*.

```
>>> pitchtools.NamedIntervalClass.from_pitch_carriers(
...     NamedPitch(-2),
...     NamedPitch(12),
... )
NamedIntervalClass(' +M2')
```

Returns named interval-class.

Special methods

`NamedIntervalClass.__abs__()`
Absolute value of named interval-class.

Returns new named interval-class.

`NamedIntervalClass.__eq__(arg)`
True when *arg* is a named interval-class with direction number, quality string and number equal to those of this named interval-class. Otherwise false.

Returns boolean.

`NamedIntervalClass.__float__()`
Changes named interval-class to float.

Returns float.

`(AbjadObject).__format__(format_specification='')`
Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`NamedIntervalClass.__hash__()`
Hashes named interval-class.

Returns integer.

`NamedIntervalClass.__int__()`
Changes named interval-class to integer.

Returns integer.

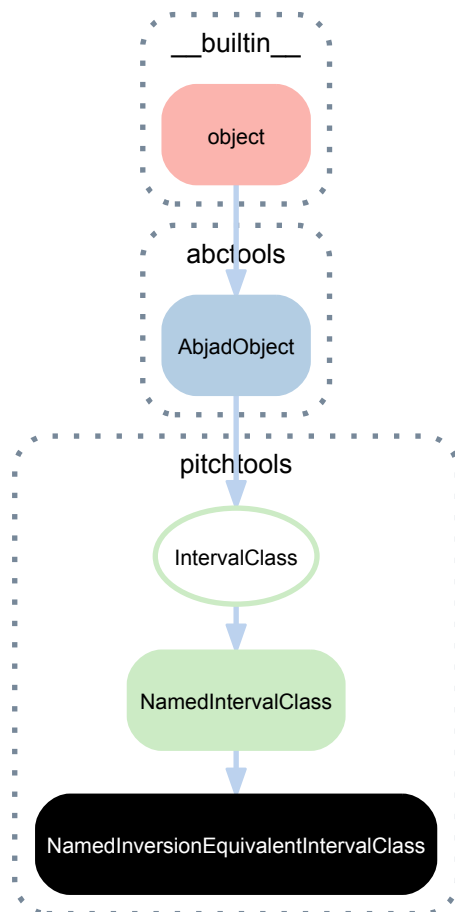
`NamedIntervalClass.__ne__(arg)`
True when named interval-class does not equal *arg*. Otherwise false.

Returns boolean.

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

`NamedIntervalClass.__str__()`
 String representation of named interval-class.
 Returns string.

13.2.10 `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`
Direction number of named interval-class.

Returns -1, 0 or 1.

`(NamedIntervalClass).direction_symbol`
Direction symbol of named interval-class.

Returns string.

`(NamedIntervalClass).direction_word`
Direction word of named interval-class.

Returns string.

`(IntervalClass).number`
Number of interval-class.

Returns number.

`(NamedIntervalClass).quality_string`
Quality string of named interval-class.

Returns string.

Class methods

`NamedInversionEquivalentIntervalClass.from_pitch_carriers` (*pitch_carrier_1*,
pitch_carrier_2)
Makes named inversion-equivalent interval-class from *pitch_carrier_1* and *pitch_carrier_2*.

```
>>> pitchtools.NamedInversionEquivalentIntervalClass.from_pitch_carriers(
...     NamedPitch(-2),
...     NamedPitch(12),
... )
NamedInversionEquivalentIntervalClass('+M2')
```

Returns named inversion-equivalent interval-class.

Special methods

`(NamedIntervalClass).__abs__()`
Absolute value of named interval-class.

Returns new named interval-class.

`NamedInversionEquivalentIntervalClass.__eq__(arg)`
True when *arg* is a named inversion-equivalent interval-class with quality string and number equal to those of this named inversion-equivalent interval-class. Otherwise false.

Returns boolean.

`(NamedIntervalClass).__float__()`
Changes named interval-class to float.

Returns float.

`(AbjadObject).__format__(format_specification='')`
Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(NamedIntervalClass).**__hash__**()
Hashes named interval-class.
Returns integer.

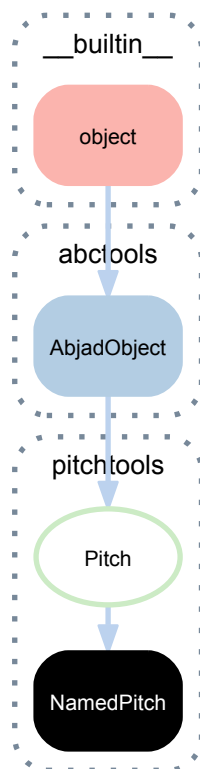
(NamedIntervalClass).**__int__**()
Changes named interval-class to integer.
Returns integer.

NamedInversionEquivalentIntervalClass.**__ne__**(arg)
True when named inversion-equivalent interval-class does not equal *arg*. Otherwise false.
Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

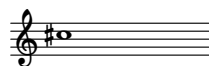
(NamedIntervalClass).**__str__**()
String representation of named interval-class.
Returns string.

13.2.11 pitchtools.NamedPitch



class pitchtools.**NamedPitch**(*args)
A named pitch.

```
>>> pitch = NamedPitch("cs'")
>>> show(pitch)
```



Bases

- `pitchtools.Pitch`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`NamedPitch.accidental`
Accidental of named pitch.

```
>>> NamedPitch("cs'").accidental
Accidental('s')
```

Returns accidental.

`(Pitch).accidental_spelling`
Accidental spelling of Abjad session.

```
>>> NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

`NamedPitch.alteration_in_semitones`
Alteration of named pitch in semitones.

```
>>> NamedPitch("cs'").alteration_in_semitones
1
```

Returns integer or float.

`NamedPitch.diatonic_pitch_class_name`
Diatonic pitch-class name of named pitch.

```
>>> NamedPitch("cs'").diatonic_pitch_class_name
'c'
```

Returns string.

`NamedPitch.diatonic_pitch_class_number`
Diatonic pitch-class number of named pitch.

```
>>> NamedPitch("cs'").diatonic_pitch_class_number
0
```

Returns integer.

`NamedPitch.diatonic_pitch_name`
Diatonic pitch name of named pitch.

```
>>> NamedPitch("cs'").diatonic_pitch_name
"c' "
```

Returns string.

`NamedPitch.diatonic_pitch_number`
Diatonic pitch number of named pitch.

```
>>> NamedPitch("cs'").diatonic_pitch_number
7
```

Returns integer.

`NamedPitch.named_pitch`
Named pitch.

```
>>> NamedPitch("cs' ").named_pitch
NamedPitch("cs' ")
```

Returns new named pitch.

NamedPitch.named_pitch_class
Named pitch-class of named pitch.

```
>>> NamedPitch("cs' ").named_pitch_class
NamedPitchClass('cs')
```

Returns named pitch-class.

NamedPitch.numbered_pitch
Numbered pitch corresponding to named pitch.

```
>>> NamedPitch("cs' ").numbered_pitch
NumberedPitch(13)
```

Returns numbered pitch.

NamedPitch.numbered_pitch_class
Numbered pitch-class corresponding to named pitch.

```
>>> NamedPitch("cs' ").numbered_pitch_class
NumberedPitchClass(1)
```

Returns numbered pitch-class.

NamedPitch.octave
Octave of named pitch.

```
>>> NamedPitch("cs' ").octave
Octave(5)
```

Returns octave.

NamedPitch.octave_number
Integer octave number of named pitch.

```
>>> NamedPitch("cs' ").octave_number
5
```

Returns integer.

NamedPitch.pitch_class_name
Pitch-class name of named pitch.

```
>>> NamedPitch("cs' ").pitch_class_name
'cs'
```

Returns string.

NamedPitch.pitch_class_number
Pitch-class number of named pitch.

```
>>> NamedPitch("cs' ").pitch_class_number
1
```

Returns integer or float.

NamedPitch.pitch_class_octave_label
Pitch-class / octave label of named pitch.

```
>>> NamedPitch("cs' ").pitch_class_octave_label
'C#5'
```

Returns string.

`NamedPitch.pitch_name`
Pitch name of named pitch.

```
>>> NamedPitch("cs' ").pitch_name
"cs' "
```

Returns string.

`NamedPitch.pitch_number`
Pitch-class number of named pitch.

```
>>> NamedPitch("cs' ").pitch_number
13
```

```
>>> NamedPitch("cff' ").pitch_number
10
```

Returns integer or float.

Methods

`NamedPitch.apply_accidental (accidental=None)`
Applies *accidental* to named pitch.

```
>>> NamedPitch("cs' ").apply_accidental('s')
NamedPitch("css' ")
```

Returns new named pitch.

`NamedPitch.invert (axis=None)`
Inverts named pitch around *axis*.

Not yet implemented.

`NamedPitch.multiply (n=1)`
Multiply pitch-class of named pitch by *n* while maintaining octave of named pitch.

```
>>> NamedPitch('d,').multiply(3)
NamedPitch('fs,')
```

Returns new named pitch.

`NamedPitch.respell_with_flats ()`
Respells named pitch with flats.

```
>>> NamedPitch("cs' ").respell_with_flats()
NamedPitch("df' ")
```

Returns new named pitch.

`NamedPitch.respell_with_sharps ()`
Respells named pitch with sharps.

```
>>> NamedPitch("df' ").respell_with_sharps()
NamedPitch("cs' ")
```

Returns new named pitch.

`NamedPitch.transpose (expr)`
Transposes named pitch 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 when *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__** ()
 Absolute value of named pitch.

```
>>> abs(pitch)
13
```

Returns nonnegative number.

`NamedPitch.__add__(interval)`
 Adds named pitch to *interval*.

```
>>> pitch + pitchtools.NamedInterval('+M2')
NamedPitch("ds' ' ")
```

Returns new named pitch.

`NamedPitch.__copy__(*args)`
 Copies named pitch.

```
>>> import copy
>>> copy.copy(pitch)
NamedPitch("cs' ' ")
```

Returns new named pitch.

`NamedPitch.__eq__(arg)`
 True when *arg* is a named pitch equal to this named pitch.

```
>>> pitch == NamedPitch("cs' ' ")
True
```

Otherwise false:

```
>>> pitch == NamedPitch("ds' ' ")
False
```

Returns boolean.

`NamedPitch.__float__()`
 Changes named pitch to float.

```
>>> float(pitch)
13.0
```

Returns float.

`(Pitch).__format__(format_specification='')`
 Formats pitch.
 Set *format_specification* to `'`, `'lilypond'` or `'storage'`.
 Returns string.

`NamedPitch.__ge__(arg)`
 True when named pitch is greater than or equal to *arg*. Otherwise false.
 Returns boolean.

`NamedPitch.__gt__(arg)`
 True when named pitch is greater than *arg*. Otherwise false.
 Returns boolean.

`(Pitch).__hash__()`
 Hashes pitch.
 Returns integer.

`(Pitch).__illustrate__()`
 Illustrates pitch.
 Returns LilyPond file.

`NamedPitch.__int__()`
 Changes named pitch to integer.

```
>>> int(pitch)
13
```

Returns integer.

`NamedPitch.__le__(arg)`

True when named pitch is less than or equal to *arg*. Otherwise false.

Returns boolean.

`NamedPitch.__lt__(arg)`

True when named pitch is less than *arg*. Otherwise false.

Returns boolean.

`NamedPitch.__ne__(arg)`

True when named pitch does not equal *arg*.

```
>>> NamedPitch("cs'") != NamedPitch("ds'")
True
```

Otherwise false:

```
>>> NamedPitch("cs'") != NamedPitch("cs'")
False
```

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

`NamedPitch.__str__()`

String representation of named pitch.

```
>>> str(NamedPitch("cs'"))
"cs' "
```

Returns string.

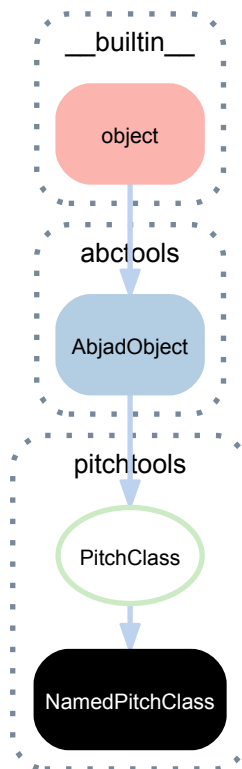
`NamedPitch.__sub__(arg)`

Subtracts *arg* from named pitch.

```
>>> NamedPitch("cs'") - NamedPitch("b'")
NamedInterval(' -M2')
```

Returns named interval.

13.2.12 pitchtools.NamedPitchClass



class pitchtools.**NamedPitchClass** (*expr=None*)

A named pitch-class.

```
>>> pitchtools.NamedPitchClass('cs')
NamedPitchClass('cs')
```

```
>>> pitchtools.NamedPitchClass(14)
NamedPitchClass('d')
```

```
>>> pitchtools.NamedPitchClass(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')
```

```
>>> pitch_class = pitchtools.NamedPitchClass('cs')
```

Bases

- pitchtools.PitchClass
- abctools.AbjadObject
- __builtin__.object

Read-only properties

`NamedPitchClass.accidental`
Accidental of named pitch-class.

```
>>> pitchtools.NamedPitchClass('cs').accidental
Accidental('s')
```

Returns accidental.

`(PitchClass).accidental_spelling`
Accidental spelling of pitch-class.

Returns string.

`NamedPitchClass.alteration_in_semitones`
Alteration of named pitch-class in semitones.

```
>>> pitchtools.NamedPitchClass('cs').alteration_in_semitones
1
```

Returns integer or float.

`NamedPitchClass.diatonic_pitch_class_name`
Diatonic pitch-class name of named interval.

```
>>> pitchtools.NamedPitchClass('cs').diatonic_pitch_class_name
'c'
```

Returns string.

`NamedPitchClass.diatonic_pitch_class_number`
Diatonic pitch-class number of named pitch-class.

```
>>> 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 new named pitch-class.

`NamedPitchClass.numbered_pitch_class`
Numbered pitch-class corresponding to named pitch-class.

```
>>> pitchtools.NamedPitchClass('cs').numbered_pitch_class
NumberedPitchClass(1)
```

Returns numbered pitch-class.

`NamedPitchClass.pitch_class_label`
Pitch-class label of named pitch-class.

```
>>> pitchtools.NamedPitchClass('cs').pitch_class_label
'C#'
```

Returns string.

`NamedPitchClass.pitch_class_name`
Pitch-class name of named pitch-class.

```
>>> pitchtools.NamedPitchClass('cs').pitch_class_name
'cs'
```

Returns string.

`NamedPitchClass.pitch_class_number`

Pitch-class number of named pitch-class.

```
>>> pitchtools.NamedPitchClass('cs').pitch_class_number
1
```

Returns integer or float.

Methods

`NamedPitchClass.apply_accidental(accidental)`

Applies *accidental* to named pitch-class.

```
>>> pitchtools.NamedPitchClass('cs').apply_accidental('qs')
NamedPitchClass('ctqs')
```

Returns new named pitch-class.

`NamedPitchClass.invert()`

Inverts named pitch-class.

Not yet implemented.

`NamedPitchClass.multiply(n=1)`

Multiplies named pitch-class by *n*.

```
>>> pitchtools.NamedPitchClass('cs').multiply(3)
NamedPitchClass('ef')
```

Returns new named pitch-class.

`NamedPitchClass.transpose(n)`

Transposes named pitch-class by named interval *n*.

```
>>> named_interval = pitchtools.NamedInterval('major', 2)
>>> pitchtools.NamedPitchClass('cs').transpose(named_interval)
NamedPitchClass('ds')
```

Returns 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__()`
Absolute value of named pitch-class.

```
>>> abs(pitch_class)
1
```

Returns nonnegative number.

`NamedPitchClass.__add__(named_interval)`
Adds *named_interval* to named pitch-class.

```
>>> pitch_class + pitchtools.NamedInterval('+M9')
NamedPitchClass('ds')
```

Return new named pitch-class.

`NamedPitchClass.__copy__(*args)`
Copies named pitch-class.

```
>>> import copy
>>> copy.copy(pitch_class)
NamedPitchClass('cs')
```

Returns new named pitch-class.

`NamedPitchClass.__eq__(expr)`
True when *expr* can be coerced to a named pitch-class with pitch-class name equal to that of this named pitch-class.

```
>>> pitch_class == 'cs'
True
```

Otherwise false:

```
>>> pitch_class == 'ds'
False
```

Returns boolean.

`NamedPitchClass.__float__()`
Changes named pitch-class to a float.

```
>>> float(pitch_class)
1.0
```

Returns float.

`(PitchClass).__format__(format_specification='')`
Formats component.

Set *format_specification* to *'*, *'lilypond'* or *'storage'*.

Returns string.

(PitchClass).**__hash__**()

Hases pitch-class.

Returns integer.

NamedPitchClass.**__int__**()

Changes named pitch-class to an integer.

```
>>> int(pitch_class)
1
```

Returns nonnegative integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

NamedPitchClass.**__str__**()

String representation of named pitch-class.

```
>>> str(pitch_class)
'cs'
```

Returns string.

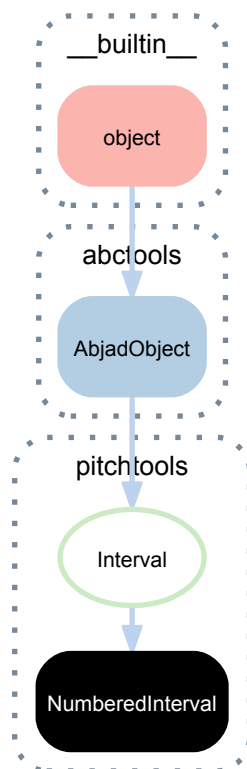
NamedPitchClass.**__sub__**(*arg*)

Subtracts *arg* from named pitch-class.

```
>>> pitch_class - pitchtools.NamedPitchClass('g')
NamedInversionEquivalentIntervalClass('+aug4')
```

Returns named inversion-equivalent interval-class.

13.2.13 pitchtools.NumberedInterval



class `pitchtools.NumberedInterval` (*arg=None*)
 A numbered interval.

```
>>> numbered_interval = pitchtools.NumberedInterval(-14)
>>> numbered_interval
NumberedInterval(-14)
```

Bases

- `pitchtools.Interval`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(Interval).cents`
 Cents of interval.

Returns nonnegative number.

`NumberedInterval.direction_number`
 Direction sign of numbered interval.

```
>>> pitchtools.NumberedInterval(-14).direction_number
-1
```

Returns integer.

`(Interval).direction_string`
 Direction string of interval.

Returns string.

`(Interval).interval_class`
Interval class of interval.

`NumberedInterval.number`
Number of numbered interval.
Returns number.

`NumberedInterval.numbered_interval_number`
Number of numbered interval.

```
>>> pitchtools.NumberedInterval(-14).numbered_interval_number
-14
```

Returns integer or float.

`NumberedInterval.semitones`
Semitones corresponding to numbered interval.
Returns nonnegative number.

Class methods

`NumberedInterval.from_pitch_carriers(pitch_carrier_1, pitch_carrier_2)`
Makes numbered interval from *pitch_carrier_1* and *pitch_carrier_2*.

```
>>> pitchtools.NumberedInterval.from_pitch_carriers(
...     NamedPitch(-2),
...     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__()`
Absolute value of numbered interval.
Returns new numbered interval.

`NumberedInterval.__add__(arg)`

Adds *arg* to numbered interval.

Returns new numbered interval.

`NumberedInterval.__copy__()`

Copies numbered interval.

Returns new numbered interval.

`NumberedInterval.__eq__(arg)`

True when *arg* is a numbered interval with number equal to that of this numbered interval. Otherwise false.

Returns boolean.

`NumberedInterval.__float__()`

Changes numbered interval to float.

Returns float.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`NumberedInterval.__ge__(other)`

$x.__ge__(y) \iff x \geq y$

`NumberedInterval.__gt__(other)`

$x.__gt__(y) \iff x > y$

`NumberedInterval.__hash__()`

Hashes numbered interval.

Returns integer.

`NumberedInterval.__int__()`

Changes numbered interval to integer.

Returns integer.

`NumberedInterval.__le__(other)`

$x.__le__(y) \iff x \leq y$

`NumberedInterval.__lt__(arg)`

True when *arg* is a numbered interval with same direction number as this numbered interval and with number greater than that of this numbered interval. Otherwise false.

Returns boolean.

`(Interval).__ne__(arg)`

True when interval does not equal *arg*.

Returns boolean.

`NumberedInterval.__neg__()`

Negates numbered interval.

Returns new numbered interval.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

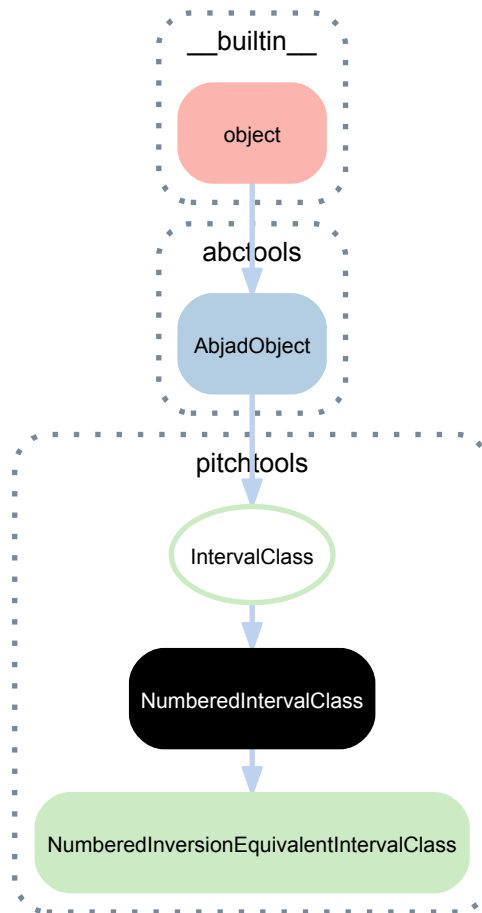
`NumberedInterval.__str__()`

String representation of numbered interval.

Returns string.

`NumberedInterval.__sub__(arg)`
 Subtracts *arg* from numbered interval.
 Returns new numbered interval.

13.2.14 `pitchtools.NumberedIntervalClass`



class `pitchtools.NumberedIntervalClass` (*token=None*)
 A numbered interval-class.

```
>>> pitchtools.NumberedIntervalClass(-14)
NumberedIntervalClass(-2)
```

Bases

- `pitchtools.IntervalClass`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`NumberedIntervalClass.direction_number`
 Direction number of numbered interval-class.
 Returns -1, 0 or 1.

`NumberedIntervalClass.direction_symbol`

Direction symbol of numbered interval class.

Returns string.

`NumberedIntervalClass.direction_word`

Direction word of numbered interval-class.

Returns string.

`(IntervalClass).number`

Number of interval-class.

Returns number.

Class methods

`NumberedIntervalClass.from_pitch_carriers` (*pitch_carrier_1*, *pitch_carrier_2*)

Makes numbered interval-class from *pitch_carrier_1* and *pitch_carrier_2*.

```
>>> pitchtools.NumberedIntervalClass.from_pitch_carriers(
...     NamedPitch(-2),
...     NamedPitch(12),
... )
NumberedIntervalClass(2)
```

Returns numbered interval-class.

Special methods

`NumberedIntervalClass.__abs__()`

Absolute value of numbered interval-class.

Returns new numbered interval-class.

`NumberedIntervalClass.__eq__(arg)`

True when *arg* is a numbered interval-class with number equal to that of this numbered interval-class. Otherwise false.

Returns boolean.

`NumberedIntervalClass.__float__()`

Changes numbered interval-class to float.

Returns float.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(IntervalClass).__hash__()`

Hashes interval-class.

Returns integer.

`NumberedIntervalClass.__int__()`

Changes numbered interval-class to integer.

Returns integer.

`(AbjadObject).__ne__(expr)`

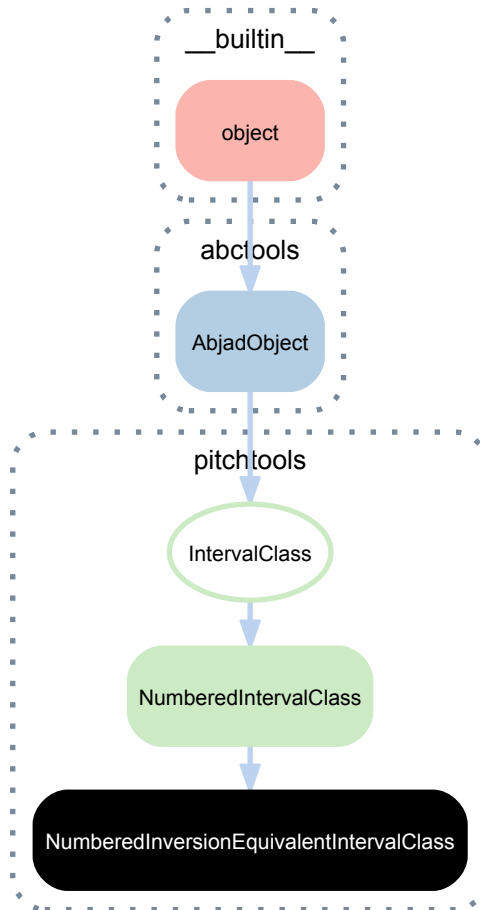
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

`(IntervalClass).__str__()`
 String representation of interval-class.
 Returns string.

13.2.15 `pitchtools.NumberedInversionEquivalentIntervalClass`



class `pitchtools.NumberedInversionEquivalentIntervalClass` (*interval_class_token=None*)
 A numbered inversion-equivalent interval-class.

```
>>> pitchtools.NumberedInversionEquivalentIntervalClass(1)
NumberedInversionEquivalentIntervalClass(1)
```

Bases

- `pitchtools.NumberedIntervalClass`
- `pitchtools.IntervalClass`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(NumberedIntervalClass).direction_number`
Direction number of numbered interval-class.

Returns -1, 0 or 1.

`(NumberedIntervalClass).direction_symbol`
Direction symbol of numbered interval class.

Returns string.

`(NumberedIntervalClass).direction_word`
Direction word of numbered interval-class.

Returns string.

`(IntervalClass).number`
Number of interval-class.

Returns number.

Class methods

`(NumberedIntervalClass).from_pitch_carriers(pitch_carrier_1, pitch_carrier_2)`
Makes numbered interval-class from *pitch_carrier_1* and *pitch_carrier_2*.

```
>>> pitchtools.NumberedIntervalClass.from_pitch_carriers(  
...     NamedPitch(-2),  
...     NamedPitch(12),  
... )  
NumberedIntervalClass(2)
```

Returns numbered interval-class.

Special methods

`NumberedInversionEquivalentIntervalClass.__abs__()`
Absolute value of numbered inversion-equivalent interval-class.

Returns new numbered inversion-equivalent interval-class.

`NumberedInversionEquivalentIntervalClass.__copy__()`
Copies numbered inversion-equivalent interval-class.

Returns new numbered inversion-equivalent interval-class.

`NumberedInversionEquivalentIntervalClass.__eq__(arg)`
True when *arg* is a numbered inversion-equivalent interval-class with number equal to that of this numbered inversion-equivalent interval-class. Otherwise false.

Returns boolean.

`(NumberedIntervalClass).__float__()`
Changes numbered interval-class to float.

Returns float.

`(AbjadObject).__format__(format_specification='')`
Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`NumberedInversionEquivalentIntervalClass.__hash__()`

Hashes numbered inversion-equivalent interval-class.

Returns integer.

`(NumberedIntervalClass).__int__()`

Changes numbered interval-class to integer.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`NumberedInversionEquivalentIntervalClass.__neg__()`

Negates numbered inversion-equivalent interval-class.

Returns new numbered inversion-equivalent interval-class.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

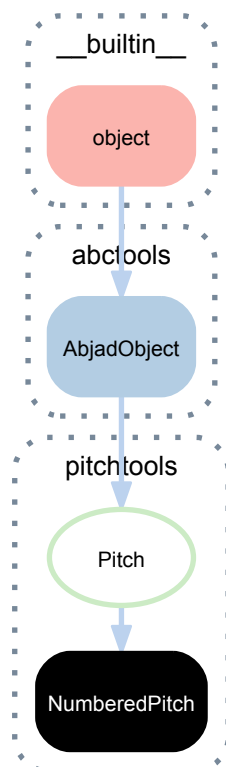
Returns string.

`NumberedInversionEquivalentIntervalClass.__str__()`

String representation of numbered inversion-equivalent interval-class.

Returns string.

13.2.16 pitchtools.NumberedPitch

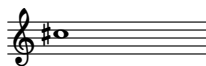


class `pitchtools.NumberedPitch` (*expr=None*)

A numbered pitch.

```
>>> numbered_pitch = pitchtools.NumberedPitch(13)
>>> numbered_pitch
NumberedPitch(13)
```

```
>>> show(numbered_pitch)
```



Bases

- `pitchtools.Pitch`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`NumberedPitch.accidental`
Accidental of numbered pitch.

```
>>> pitchtools.NumberedPitchClass(13).accidental
Accidental('s')
```

Returns accidental.

`(Pitch).accidental_spelling`
Accidental spelling of Abjad session.

```
>>> NamedPitch("c").accidental_spelling
'mixed'
```

Returns string.

`NumberedPitch.alteration_in_semitones`
Alteration of numbered pitch in semitones.

```
>>> pitchtools.NumberedPitchClass(13).alteration_in_semitones
1
```

Returns integer or float.

`NumberedPitch.diatonic_pitch_class_name`
Diatonic pitch-class name corresponding to numbered pitch.

```
>>> pitchtools.NumberedPitch(13).diatonic_pitch_class_name
'c'
```

Returns string.

`NumberedPitch.diatonic_pitch_class_number`
Diatonic pitch-class number of numbered pitch.

```
>>> pitchtools.NumberedPitch(13).diatonic_pitch_class_number
0
```

Returns integer.

`NumberedPitch.diatonic_pitch_name`
Diatonic pitch name of numbered pitch.

```
>>> pitchtools.NumberedPitch(13).diatonic_pitch_name
'c' / ''
```

Returns string.

`NumberedPitch.diatonic_pitch_number`
Diatonic pitch-class number corresponding to numbered pitch.


```
>>> pitchtools.NumberedPitch(13).diatonic_pitch_number
7
```

Returns integer.

NumberedPitch.named_pitch

Named pitch corresponding to numbered pitch.

```
>>> pitchtools.NumberedPitch(13).named_pitch
NamedPitch("cs' ' ")
```

Returns named pitch.

NumberedPitch.named_pitch_class

Named pitch-class corresponding to numbered pitch.

```
>>> 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 new numbered pitch.

NumberedPitch.numbered_pitch_class

Numbered pitch-class corresponding to numbered pitch.

```
>>> pitchtools.NumberedPitch(13).numbered_pitch_class
NumberedPitchClass(1)
```

Returns numbered pitch-class.

NumberedPitch.octave

Octave of numbered pitch.

```
>>> pitchtools.NumberedPitch(13).octave
Octave(5)
```

Returns octave.

NumberedPitch.octave_number

Octave number of numbered pitch.

```
>>> pitchtools.NumberedPitch(13).octave_number
5
```

Returns integer.

NumberedPitch.pitch_class_name

Pitch-class name of numbered pitch.

```
>>> pitchtools.NumberedPitch(13).pitch_class_name
'cs'
```

Returns string.

NumberedPitch.pitch_class_number

Pitch-class number of numbered pitch.

```
>>> pitchtools.NumberedPitch(13).pitch_class_number
1
```

Returns integer or float.

`NumberedPitch.pitch_class_octave_label`

Pitch-class / octave label of numbered pitch.

```
>>> pitchtools.NumberedPitch(13).pitch_class_octave_label
'C#5'
```

Returns string.

`NumberedPitch.pitch_name`

Pitch name corresponding to numbered pitch.

```
>>> pitchtools.NumberedPitch(13).pitch_name
'c#' "
```

Returns string.

`NumberedPitch.pitch_number`

Pitch number of numbered pitch.

```
>>> pitchtools.NumberedPitch(13).pitch_number
13
```

Returns number.

Methods

`NumberedPitch.apply_accidental (accidental=None)`

Applies *accidental* to numbered pitch.

```
>>> pitchtools.NumberedPitch(13).apply_accidental('flat')
NumberedPitch(12)
```

Returns new numbered pitch.

`NumberedPitch.invert (axis=None)`

Inverts numbered pitch around *axis*.

Not yet implemented.

`NumberedPitch.multiply (n=1)`

Multiplies pitch-class of numbered pitch by *n* and maintains octave.

```
>>> pitchtools.NumberedPitch(14).multiply(3)
NumberedPitch(18)
```

Returns new numbered pitch.

`NumberedPitch.transpose (n=0)`

Transposes numbered pitch by *n* semitones.

```
>>> pitchtools.NumberedPitch(13).transpose(1)
NumberedPitch(14)
```

Returns 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
```

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 when *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__()`
 Absolute value of numbered pitch.

Returns pitch number.

`NumberedPitch.__add__(arg)`
 Adds *arg* to numberd pitch.

Returns new numbered pitch.

`NumberedPitch.__eq__(arg)`
 True when *arg* is a numbered pitch with pitch number equal to that of this numbered pitch. Otherwise false.

Returns boolean.

`NumberedPitch.__float__()`

Changes numbered pitch to float.

Returns float.

`(Pitch).__format__(format_specification='')`

Formats pitch.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

`NumberedPitch.__ge__(other)`

`x.__ge__(y) <==> x>=y`

`NumberedPitch.__gt__(other)`

`x.__gt__(y) <==> x>y`

`NumberedPitch.__hash__()`

Hashes numbered pitch.

Returns integer.

`(Pitch).__illustrate__()`

Illustrates pitch.

Returns LilyPond file.

`NumberedPitch.__int__()`

Changes numbered pitch to integer.

Returns integer.

`NumberedPitch.__le__(other)`

`x.__le__(y) <==> x<=y`

`NumberedPitch.__lt__(arg)`

True when *arg* is a numbered pitch with pitch number greater than that of this numbered pitch. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`NumberedPitch.__neg__()`

Negates numbered pitch.

Returns new numbered pitch.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

`NumberedPitch.__str__()`

String representation of numbered pitch.

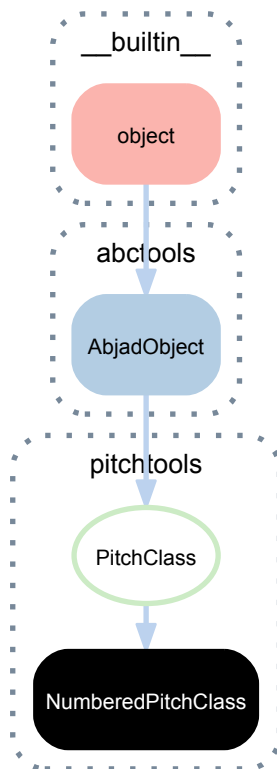
Returns string.

`NumberedPitch.__sub__(arg)`

Subtracts *arg* from numbered pitch.

Returns numbered interval.

13.2.17 `pitchtools.NumberedPitchClass`



class `pitchtools.NumberedPitchClass` (*expr=None*)
 A numbered pitch-class.

```
>>> pitchtools.NumberedPitchClass(13)
NumberedPitchClass(1)
```

```
>>> pitchtools.NumberedPitchClass('d')
NumberedPitchClass(2)
```

```
>>> pitchtools.NumberedPitchClass(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)
```

Bases

- `pitchtools.PitchClass`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`NumberedPitchClass.accidental`
Accidental of numbered pitch-class.

```
>>> pitchtools.NumberedPitchClass(1).accidental
Accidental('s')
```

Returns accidental.

`(PitchClass).accidental_spelling`
Accidental spelling of pitch-class.

Returns string.

`NumberedPitchClass.alteration_in_semitones`
Alteration of numbered pitch-class 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 corresponding to numbered pitch-class.

```
>>> pitchtools.NumberedPitchClass(1).diatonic_pitch_class_name
'c'
```

Returns string.

`NumberedPitchClass.diatonic_pitch_class_number`
Diatonic pitch-class number corresponding to numbered pitch-class.

```
>>> pitchtools.NumberedPitchClass(1).diatonic_pitch_class_number
0
```

Returns integer.

`NumberedPitchClass.named_pitch_class`
Named pitch-class corresponding to numbered 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 new numbered pitch-class.

`NumberedPitchClass.pitch_class_label`
Pitch-class / octave label of numbered pitch-class.

```
>>> 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 number.

Methods

`NumberedPitchClass.apply_accidental (accidental=None)`
Applies *accidental* to numbered pitch-class.

```
>>> pitchtools.NumberedPitchClass(1).apply_accidental('flat')
NumberedPitchClass(0)
```

Returns new numbered pitch-class.

`NumberedPitchClass.invert ()`
Invertes numbered pitch-class.

Returns new numbered pitch-class.

`NumberedPitchClass.multiply (n=1)`
Multiplies pitch-class number by *n*.

```
>>> pitchtools.NumberedPitchClass(11).multiply(3)
NumberedPitchClass(9)
```

Returns new numbered pitch-class.

`NumberedPitchClass.transpose (n)`
Transposes numbered pitch-class by *n* semitones.

Returns 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__()`
Absolute value of numbered pitch-class.

```
>>> pitch_class = pitchtools.NumberedPitchClass(9)
>>> abs(pitch_class)
9
```

Returns nonnegative number.

`NumberedPitchClass.__add__(expr)`
Adds *expr* to numbered pitch-class.

```
>>> pitch_class = pitchtools.NumberedPitchClass(9)
>>> interval = pitchtools.NumberedInterval(4)
>>> pitch_class + interval
NumberedPitchClass(1)
```

Returns new numbered pitch-class.

`NumberedPitchClass.__copy__(*args)`
Copies numbered pitch-class.

```
>>> import copy
>>> pitch_class = pitchtools.NumberedPitchClass(9)
>>> copy.copy(pitch_class)
NumberedPitchClass(9)
```

Returns new numbered pitch-class.

`NumberedPitchClass.__eq__(arg)`
True when *arg* is a numbered pitch-class with pitch-class number equal to that of this numbered pitch-class.

```
>>> pitch_class_1 = pitchtools.NumberedPitchClass(9)
>>> pitch_class_2 = pitchtools.NumberedPitchClass(3)
>>> pitch_class_1 == pitch_class_1
True
```

Otherwise false:

```
>>> pitch_class_1 == pitch_class_2
False
```

Returns boolean.

`NumberedPitchClass.__float__()`
Changes numbered pitch-class to float.


```
>>> pitch_class = pitchtools.NumberedPitchClass(9)
>>> float(pitch_class)
9.0
```

Returns float.

(PitchClass).**__format__**(*format_specification*='')

Formats component.

Set *format_specification* to '', 'lilypond' or 'storage'.

Returns string.

(PitchClass).**__hash__**()

Hases pitch-class.

Returns integer.

NumberedPitchClass.**__int__**()

Changes numbered pitch-class to integer.

```
>>> pitch_class = pitchtools.NumberedPitchClass(9)
>>> int(pitch_class)
9
```

Returns integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

NumberedPitchClass.**__neg__**()

Negates numbered pitch-class.

```
>>> pitch_class = pitchtools.NumberedPitchClass(9)
>>> -pitch_class
NumberedPitchClass(3)
```

Returns new numbered pitch-class.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

NumberedPitchClass.**__str__**()

String representation of numbered pitch-class.

Returns string.

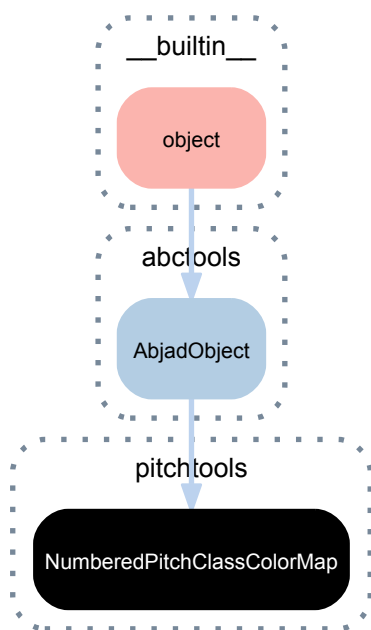
NumberedPitchClass.**__sub__**(*expr*)

Subtracts *expr* from numbered pitch-class.

Subtraction defined against both numbered intervals and against other pitch-classes.

Returns numbered inversion-equivalent interval-class.

13.2.18 `pitchtools.NumberedPitchClassColorMap`



class `pitchtools.NumberedPitchClassColorMap` (*pitch_iterables=None, colors=None*)
 A numbered pitch-class color map.

```

>>> pitches = [
...     [-8, 2, 10, 21],
...     [0, 11, 32, 41],
...     [15, 25, 42, 43],
... ]
>>> colors = ['red', 'green', 'blue']
>>> color_map = pitchtools.NumberedPitchClassColorMap(pitches, colors)
  
```

Numbered pitch-class color maps are immutable.

Bases

- `abctools.AbjadObject`
- `___builtin___object`

Read-only properties

`NumberedPitchClassColorMap.colors`
 Colors of color map.

```

>>> color_map.colors
['red', 'green', 'blue']
  
```

Returns list.

`NumberedPitchClassColorMap.is_twelve_tone_complete`
 True when color map contains all 12-ET pitch-classes.

```

>>> color_map.is_twelve_tone_complete
True
  
```

Return boolean.

`NumberedPitchClassColorMap.is_twenty_four_tone_complete`
 True when color map contains all 24-ET pitch-classes.

```
>>> color_map.is_twenty_four_tone_complete
False
```

Return boolean.

NumberedPitchClassColorMap.**pairs**
Pairs of color map.

```
>>> for pair in color_map.pairs:
...     pair
(0, 'green')
(1, 'blue')
(2, 'red')
(3, 'blue')
(4, 'red')
(5, 'green')
(6, 'blue')
(7, 'blue')
(8, 'green')
(9, 'red')
(10, 'red')
(11, 'green')
```

Returns list.

NumberedPitchClassColorMap.**pitch_iterables**
Pitch iterables of color map.

```
>>> color_map.pitch_iterables
[[-8, 2, 10, 21], [0, 11, 32, 41], [15, 25, 42, 43]]
```

Returns ?

Methods

NumberedPitchClassColorMap.**get** (*key*, *alternative=None*)
Gets *key* from color map.

```
>>> color_map.get(11)
'green'
```

Returns *alternative* when *key* is not found.

Returns string.

Special methods

(AbjadObject).**__eq__** (*expr*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__** (*format_specification=''*)
Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

NumberedPitchClassColorMap.**__getitem__** (*pc*)
Gets color corresponding to *pc* in color map.

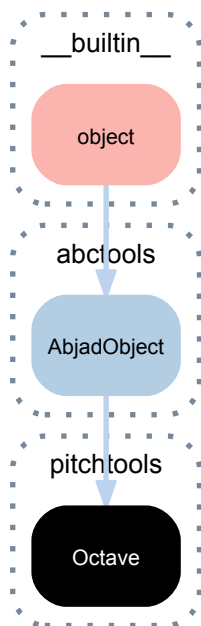
```
>>> color_map[11]
'green'
```

Returns string.

(AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

13.2.19 pitchtools.Octave



class pitchtools.**Octave**(*octave_number=None*)
An octave.

```
>>> pitchtools.Octave(4)
Octave(4)
```

```
>>> pitchtools.Octave(",", " ")
Octave(1)
```

```
>>> pitchtools.Octave(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*)

Makes octave from *pitch_name*.

```
>>> pitchtools.Octave.from_pitch_name('cs')
Octave(3)
```

Returns integer.

Octave.**from_pitch_number**(*pitch_number*)

Makes octave from *pitch_number*.

```
>>> 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.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

Octave.**__hash__**()

Hashes octave.

Returns integer.

Octave.**__int__**()

Changes octave to integer.

```
>>> int(pitchtools.Octave(3))
3
```

Returns integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

Octave.**__str__**()

String representation of octave.

Defined equal to LilyPond octave / tick representation of octave.

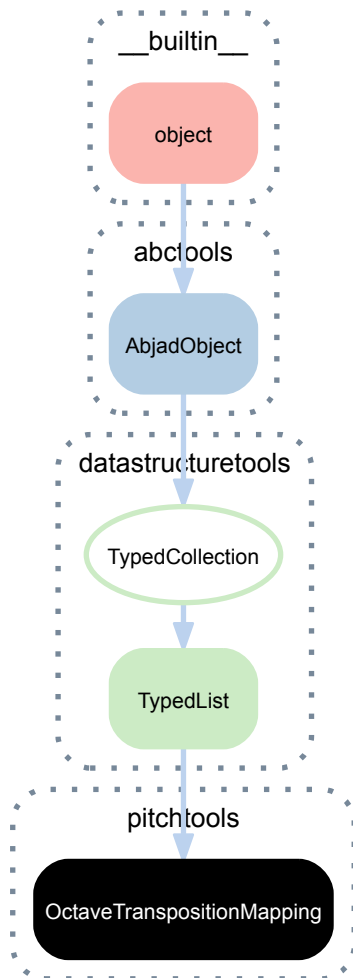
```
>>> str(pitchtools.Octave(4))
""
```

```
>>> str(pitchtools.Octave(1))
','
```

```
>>> str(pitchtools.Octave(3))
''
```

Returns string.

13.2.20 pitchtools.OctaveTranspositionMapping



```
class pitchtools.OctaveTranspositionMapping(tokens=None, item_class=None,
                                             keep_sorted=None,      cus-
                                             tom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.

Returns string or none.

`(TypedList).keep_sorted`
Sorts collection on mutation if true.

Methods

`(TypedList).append(token)`
Changes *token* to item and appends.

```
>>> 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)`
Changes *token* to item and returns 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)`
Changes *tokens* to items and extends.

```
>>> 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)`
Changes *token* to item and returns index.


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

Returns index.

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

```
>>> 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.

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

(TypedList) **.remove** (*token*)
Changes *token* to item and removes.

```
>>> 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*)
True when typed collection container *token*. Otherwise false.
Returns boolean.

(TypedList).**__delitem__**(*i*)
 Aliases list.**__delitem__**().

(TypedCollection).**__eq__**(*expr*)
 True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

OctaveTranspositionMapping.**__format__**(*format_specification*='')
 Formats octave transposition mapping.

Set *format_specification* to ' or 'storage'. Interprets ' equal to 'storage'.

```
>>> print format(mapping)
pitchtools.OctaveTranspositionMapping(
  [
    pitchtools.OctaveTranspositionMappingComponent(
      pitchtools.PitchRange(
        '[A0, C4]'
      ),
      pitchtools.NumberedPitch(15)
    ),
    pitchtools.OctaveTranspositionMappingComponent(
      pitchtools.PitchRange(
        '[C4, C8]'
      ),
      pitchtools.NumberedPitch(27)
    ),
  ]
)
```

Returns string.

(TypedList).**__getitem__**(*i*)
 Aliases list.**__getitem__**().

(TypedList).**__iadd__**(*expr*)
 Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList(
  [
    indicatorortools.Dynamic(
      'ppp'
    ),
    indicatorortools.Dynamic(
      'p'
    ),
    indicatorortools.Dynamic(
      'mp'
    ),
    indicatorortools.Dynamic(
      'mf'
    ),
    indicatorortools.Dynamic(
      'fff'
    ),
  ],
  item_class=indicatorortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()
 Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()
Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)
True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.

Returns string.

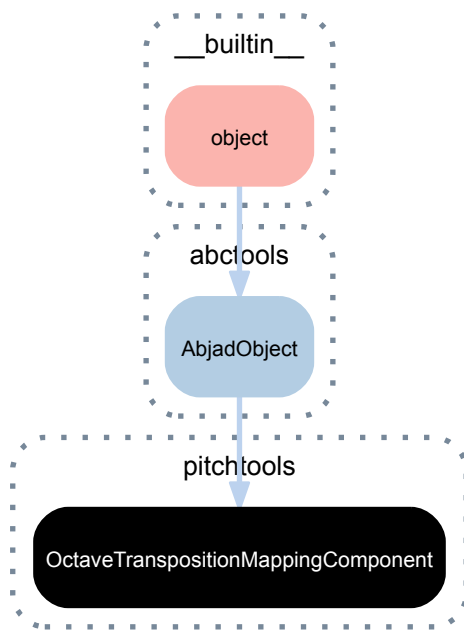
(TypedList).**__reversed__**()
Aliases list.**__reversed__**().

(TypedList).**__setitem__**(*i, expr*)
Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```

13.2.21 pitchtools.OctaveTranspositionMappingComponent



class `pitchtools.OctaveTranspositionMappingComponent` (*args)
 An octave transposition mapping component.

```
>>> mc = pitchtools.OctaveTranspositionMappingComponent(' [A0, C8]', 15)
>>> mc
OctaveTranspositionMappingComponent(' [A0, C8]', 15)
```

Initializes from input parameters separately, from a pair, from a string or from another mapping component.

Models `pitchtools.transpose_pitch_number_by_octave_transposition_mapping` input part. (See the docs for that function.)

Octave transposition mapping components are mutable.

Bases

- `abcctools.AbjadObject`
- `__builtin__.object`

Read/write properties

`OctaveTranspositionMappingComponent.source_pitch_range`
 Gets and sets source pitch range of mapping component.

```
>>> mc.source_pitch_range
PitchRange(' [A0, C8]')
```

Returns pitch range or none.

`OctaveTranspositionMappingComponent.target_octave_start_pitch`
 Gets and sets target octave start pitch of mapping component.

```
>>> mc.target_octave_start_pitch
NumberedPitch(15)
```

Returns numbered pitch or none.

Special methods

`OctaveTranspositionMappingComponent.__eq__(expr)`

True when *expr* is an octave transposition mapping component with source pitch range and target octave start pitch equal to those of this octave transposition mapping component. Otherwise false.

Returns boolean.

`OctaveTranspositionMappingComponent.__format__(format_specification='')`

Formats mapping component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

`OctaveTranspositionMappingComponent.__ne__(expr)`

True when octave transposition mapping component does not equal *expr*. Otherwise false.

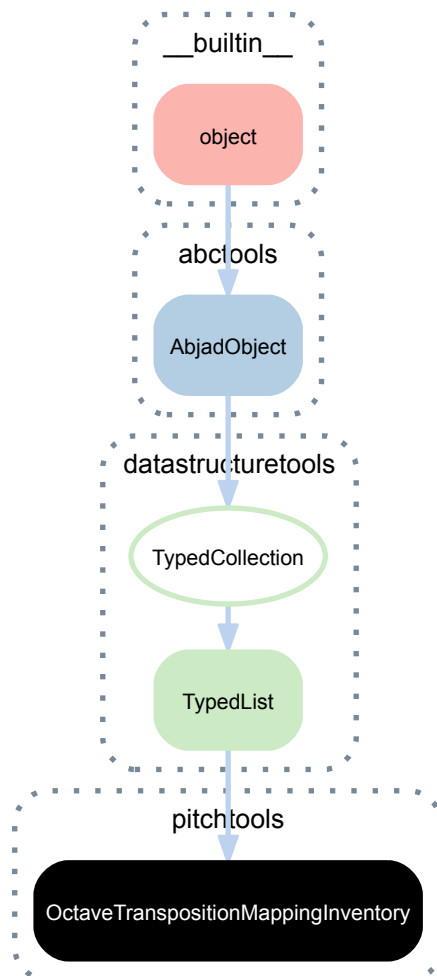
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2.22 pitchtools.OctaveTranspositionMappingInventory



```
class pitchtools.OctaveTranspositionMappingInventory (tokens=None,
                                                    item_class=None,
                                                    keep_sorted=None,      cus-
                                                    tom_identifier=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 format(inventory)
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.

Read/write properties

(`TypedCollection`).**custom_identifier**
Gets and sets custom identifier of typed collection.

Returns string or none.

(TypedList) **.keep_sorted**

Sorts collection on mutation if true.

Methods

(TypedList) **.append** (*token*)

Changes *token* to item and appends.

```
>>> 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*)

Changes *token* to item and returns 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*)

Changes *tokens* to items and extends.

```
>>> 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*)

Changes *token* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *token*)

Changes *token* to item and inserts.

```
>>> 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.

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

Aliases list.pop().

(TypedList) **.remove** (*token*)

Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) **.__format__** (*format_specification=''*)

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList) **.__getitem__** (*i*)

Aliases list.__getitem__().

(TypedList) **.__iadd__** (*expr*)

Changes tokens in *expr* to items and extends.

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



```
>>> print format(dynamic_collection)
datastructuretools.TypedList (
  [
    indicatortools.Dynamic(
      'ppp'
    ),
    indicatortools.Dynamic(
      'p'
    ),
    indicatortools.Dynamic(
      'mp'
    ),
    indicatortools.Dynamic(
      'mf'
    ),
    indicatortools.Dynamic(
      'fff'
    ),
  ],
  item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList).**__reversed__**()

Aliases list.**__reversed__**().

(TypedList).**__setitem__**(*i, expr*)

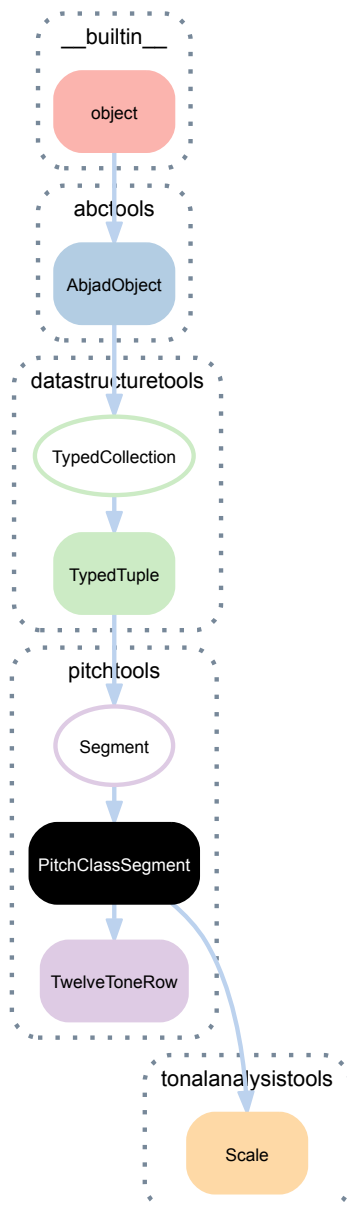
Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```

13.2.23 pitchtools.PitchClassSegment



class `pitchtools.PitchClassSegment` (*tokens=None*, *item_class=None*, *custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`

Gets and sets custom identifier of typed collection.

Returns string or none.

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])
```

Returns new pitch-class segment.

(TypedTuple) **.count** (*token*)

Changes *token* to item.

Returns count in collection.

(TypedTuple) **.index** (*token*)

Changes *token* to item.

Returns 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])
```

Returns 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 = TimeSignature((5, 4))
>>> attach(time_signature, 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])
```

Returns new pitch-class segment.

`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])
```

Returns 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'])
```

Returns 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])
```

Returns new pitch-class segment.

Class methods

`PitchClassSegment`.**from_selection**(*selection*, *item_class=None*, *custom_identifier=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*)
Adds typed tuple to *expr*.

Returns new typed tuple.

(`TypedTuple`) **__contains__**(*token*)
Change *token* to item and return true if item exists in collection.

Returns none.

(TypedCollection) .**__eq__**(*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) .**__format__**(*format_specification*='')

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedTuple) .**__getitem__**(*i*)

Gets *i* from type tuple.

Returns item.

(TypedTuple) .**__getslice__**(*start*, *stop*)

Gets slice from *start* to *stop* in typed tuple.

Returns new typed tuple.

(TypedTuple) .**__hash__**()

Hashes typed tuple.

Returns integer.

(TypedCollection) .**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedCollection) .**__makenew__**(*tokens=None*, *item_class=None*, *custom_identifier=None*)

Makes new typed collection with optional new values.

Returns new typed collection.

(TypedTuple) .**__mul__**(*expr*)

Multiplies typed tuple by *expr*.

Returns new typed tuple.

(TypedCollection) .**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedTuple) .**__rmul__**(*expr*)

Multiplies *expr* by typed tuple.

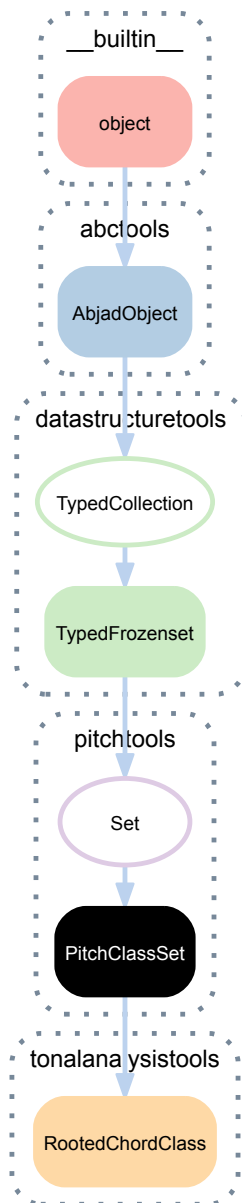
Returns new typed tuple.

(Segment) .**__str__**()

String representation of segment.

Returns string.

13.2.24 pitchtools.PitchClassSet



class `pitchtools.PitchClassSet` (*tokens=None, item_class=None, custom_identifer=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'])
```

Bases

- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

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

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

Methods

`(TypedFrozenSet).copy()`
Copies typed frozen set.
Returns new typed frozen set.

`(TypedFrozenSet).difference(expr)`
Typed frozen set set-minus *expr*.
Returns new typed frozen set.

`(TypedFrozenSet).intersection(expr)`
Set-theoretic intersection of typed frozen set and *expr*.
Returns new typed frozen set.

`PitchClassSet.invert()`
Inverts 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 pitch-class set is transposed subset of *pcset*. Otherwise false:

```
>>> 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 pitch-class set is transposed superset of *pcset*. Otherwise false:

```
>>> 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*)

True when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).issubset` (*expr*)

True when typed frozen set is a subset of *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).issuperset` (*expr*)

True when typed frozen set is a superset of *expr*. Otherwise false.

Returns boolean.

`PitchClassSet.multiply` (*n*)

Multiplies pitch-class set by *n*.

```
>>> pitchtools.PitchClassSet(
...     [-2, -1.5, 6, 7, -1.5, 7],
...     ).multiply(5)
PitchClassSet([2, 4.5, 6, 11])
```

Returns new pitch-class set.

`PitchClassSet.order_by` (*pitch_class_segment*)

Orders pitch-class set by *pitch_class_segment*.

Returns pitch-class segment.

`(TypedFrozenSet).symmetric_difference` (*expr*)

Symmetric difference of typed frozen set and *expr*.

Returns new typed frozen set.

`PitchClassSet.transpose` (*expr*)

Transposes all pitch-classes in pitch-class set by *expr*.

Returns new pitch-class set.

`(TypedFrozenSet).union` (*expr*)

Union of typed frozen set and *expr*.

Returns new typed frozen set.

Class methods

`PitchClassSet.from_selection` (*selection*, *item_class=None*, *custom_identifier=None*)

Makes pitch-class set from *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)`

Logical AND of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedCollection).__contains__(token)`

True when typed collection container *token*. Otherwise false.

Returns boolean.

`(TypedCollection).__eq__(expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TypedFrozenSet).__ge__(expr)`

True when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).__gt__(expr)`

True when typed frozen set is greater than *expr*. Otherwise false.

Returns boolean.

`PitchClassSet.__hash__()`

Hashes pitch-class set.

Returns integer.

`(TypedCollection).__iter__()`

Iterates typed collection.

Returns generator.

`(TypedFrozenSet).__le__(expr)`

True when typed frozen set is less than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedCollection).__len__()`

Length of typed collection.

Returns nonnegative integer.

`(TypedFrozenSet).__lt__(expr)`

True when typed frozen set is less than *expr*. Otherwise false.

Returns boolean.

`(TypedCollection).__makenew__(tokens=None, item_class=None, custom_identifier=None)`

Makes new typed collection with optional new values.

Returns new typed collection.

(TypedFrozenset) .**__ne__**(*expr*)
True when typed frozen set is not equal to *expr*. Otherwise false.
Returns boolean.

(TypedFrozenset) .**__or__**(*expr*)
Logical OR of typed frozen set and *expr*.
Returns new typed frozen set.

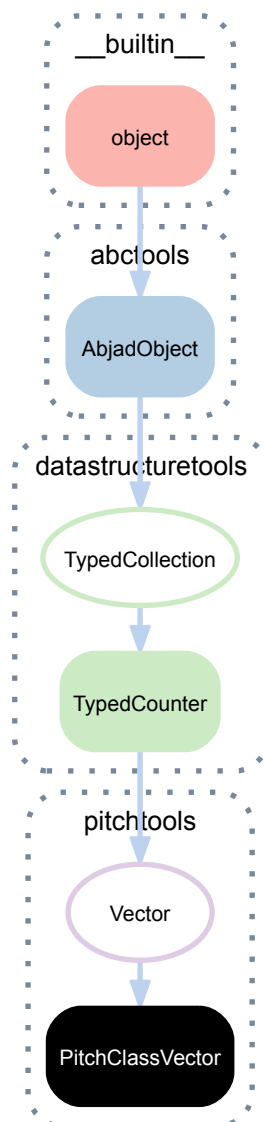
(AbjadObject) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(Set) .**__str__**()
String representation of set.
Returns string.

(TypedFrozenset) .**__sub__**(*expr*)
Subtracts *expr* from typed frozen set.
Returns new typed frozen set.

(TypedFrozenset) .**__xor__**(*expr*)
Logical XOR of typed frozen set and *expr*.
Returns new typed frozen set.

13.2.25 pitchtools.PitchClassVector



class `pitchtools.PitchClassVector` (*tokens=None, item_class=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
 Gets and sets custom identifier of typed collection.
 Returns string or none.

Methods

`(TypedCounter).clear()`
 Clears typed counter.
 Returns none.

`(TypedCounter).copy()`
 Copies typed counter.
 Returns new typed counter.

`(TypedCounter).elements()`
 Elements in typed counter.

`(TypedCounter).items()`
 Items in typed counter.
 Returns tuple.

`(TypedCounter).iteritems()`
 Iterates items in typed counter.
 Yields items.

`(TypedCounter).iterkeys()`
 Iterates keys in typed counter.

`(TypedCounter).intervalues()`
 Iterates values in typed counter.

`(TypedCounter).keys()`
 Keys in typed counter.

`(TypedCounter).most_common(n=None)`
 Please document.

`(TypedCounter).subtract(iterable=None, **kwargs)`
 Stracts *iterable* from typed counter.

`(TypedCounter).update(iterable=None, **kwargs)`
 Updates typed counter with *iterable*.

`(TypedCounter).values()`
 Values of typed counter.

`(TypedCounter).viewitems()`
 Please document.

`(TypedCounter).viewkeys()`
 Please document.

`(TypedCounter).viewvalues()`
 Please document.

Class methods

`PitchClassVector.from_selection(selection, item_class=None, custom_identifier=None)`
 Makes pitch-class vector from *selection*.

Returns pitch-class vector.

Special methods

`(TypedCounter).__add__(expr)`

Adds typed counter to *expr*.

Returns new typed counter.

`(TypedCounter).__and__(expr)`

Logical AND of typed counter and *expr*.

Returns new typed counter.

`(TypedCollection).__contains__(token)`

True when typed collection container *token*. Otherwise false.

Returns boolean.

`(TypedCounter).__delitem__(token)`

Deletes *token* from typed counter.

Returns none.

`(TypedCollection).__eq__(expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TypedCounter).__getitem__(token)`

Gets *token* from typed counter.

Returns item.

`(TypedCollection).__iter__()`

Iterates typed collection.

Returns generator.

`(TypedCollection).__len__()`

Length of typed collection.

Returns nonnegative integer.

`(TypedCollection).__makenew__(tokens=None, item_class=None, custom_identifier=None)`

Makes new typed collection with optional new values.

Returns new typed collection.

`(TypedCounter).__missing__(token)`

Returns zero.

Returns zero.

`(TypedCollection).__ne__(expr)`

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(TypedCounter).**__or__**(*expr*)
 Logical OR of typed counter and *expr*.
 Returns new typed counter.

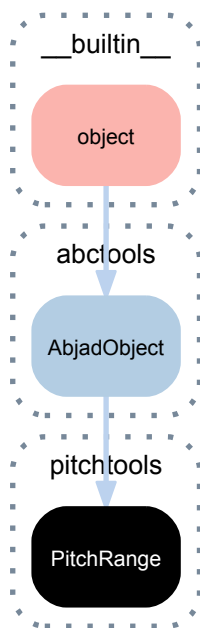
(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

(TypedCounter).**__setitem__**(*token*, *value*)
 Sets typed counter *token* to *value*.
 Returns none.

(Vector).**__str__**()
 String representation of vector.
 Returns string.

(TypedCounter).**__sub__**(*expr*)
 Subtracts *expr* from typed counter.
 Returns new typed counter.

13.2.26 pitchtools.PitchRange



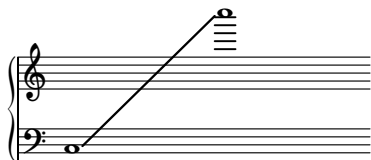
class pitchtools.**PitchRange**(**args*, ***kwargs*)
 A pitch range.

```

>>> pitch_range = pitchtools.PitchRange(-12, 36,
...     pitch_range_name='four-octave range')
>>> print format(pitch_range)
pitchtools.PitchRange(
    '[C3, C7]',
    pitch_range_name='four-octave range',
    pitch_range_name_markup=markuptools.Markup(
        ('four-octave range',)
    ),
)
  
```

```

>>> show(pitch_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 representation of pitch range.

```
>>> pitch_range.one_line_named_pitch_repr  
'[C3, C7]'
```

Returns string.

`PitchRange.one_line_numbered_pitch_repr`

One-line numbered pitch representation of pitch 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`

Pitch range name markup.

```
>>> 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 pitch range.

```
>>> pitch_range.start_pitch  
NamedPitch('c')
```

Returns pitch.

`PitchRange.start_pitch_is_included_in_range`

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 pitch range.

```
>>> pitch_range.stop_pitch
NamedPitch("c'""")
```

Returns pitch.

`PitchRange.stop_pitch_is_included_in_range`

True when stop pitch is included in range. Otherwise false:

```
>>> pitch_range.stop_pitch_is_included_in_range
True
```

Returns boolean.

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 pitches enclosed in some combination of square brackets and round parentheses.

Returns boolean.

Special methods

`PitchRange.__contains__(arg)`

True when pitch range contains *arg*. Otherwise false.

Returns boolean.

`PitchRange.__eq__(arg)`

True when *arg* is a pitch range with start and stop equal to those of this pitch range. Otherwise false.

Returns boolean.

`PitchRange.__format__(format_specification='')`

Formats pitch range.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`PitchRange.__ge__(arg)`

True when start pitch of pitch range is greater than or equal to *arg*. Otherwise false.

Returns boolean.

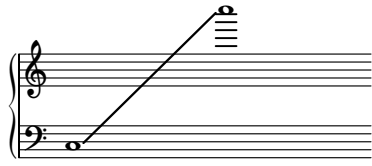
`PitchRange.__gt__(arg)`

True when start pitch of pitch range is greater than *arg*. Otherwise false.

Returns boolean.

`PitchRange.__illustrate__()`
Illustrates pitch range.

```
>>> show(pitch_range)
```



Returns LilyPond file.

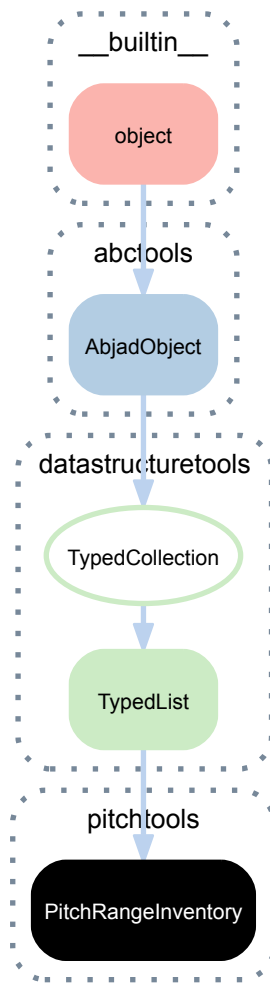
`PitchRange.__le__(arg)`
True when stop pitch of pitch-range is less than or equal to *arg*. Otherwise false.
Returns boolean.

`PitchRange.__lt__(arg)`
True when stop pitch of pitch-range is less than *arg*. Otherwise false.
Returns boolean.

`PitchRange.__ne__(arg)`
True when pitch range does not equal *arg*. Otherwise false.
Returns boolean.

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

13.2.27 pitchtools.PitchRangeInventory



class `pitchtools.PitchRangeInventory` (*tokens=None, item_class=None, keep_sorted=None, custom_identifier=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.

Read/write properties

(TypedCollection) .**custom_identifier**
Gets and sets custom identifier of typed collection.

Returns string or none.

(TypedList) .**keep_sorted**
Sorts collection on mutation if true.

Methods

(TypedList) .**append** (*token*)
Changes *token* to item and appends.

```
>>> 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*)
Changes *token* to item and returns 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*)
Changes *tokens* to items and extends.

```
>>> 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*)
Changes *token* to item and returns index.

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

Returns index.

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

```
>>> 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.

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

Aliases list.pop().

(TypedList) **.remove** (*token*)

Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) **.__format__** (*format_specification=''*)

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList) **.__getitem__** (*i*)

Aliases list.__getitem__().

(TypedList) **.__iadd__** (*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList(
[
    indicatortools.Dynamic(
        'ppp'
    ),
    indicatortools.Dynamic(
        'p'
    ),
    indicatortools.Dynamic(
        'mp'
    ),
    indicatortools.Dynamic(
        'mf'
    ),
    indicatortools.Dynamic(
        'fff'
    ),
],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, cus-*
tom_identifier=None)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList).**__reversed__**()

Aliases list.**__reversed__**().

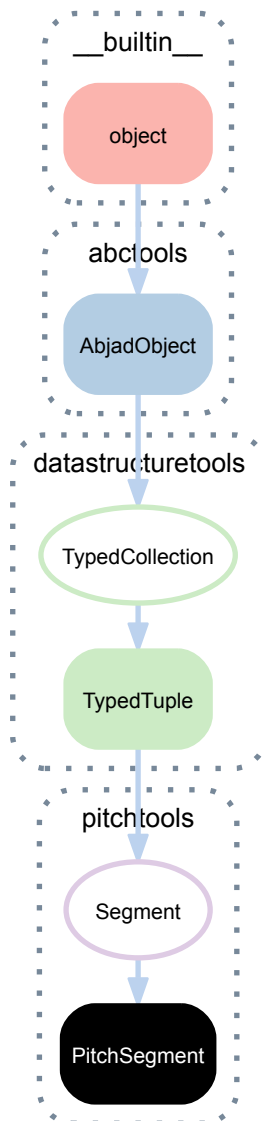
(TypedList).**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```

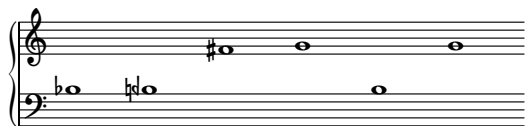
13.2.28 pitchtools.PitchSegment



class `pitchtools.PitchSegment` (*tokens=None, item_class=None, custom_identifer=None*)
 A pitch segment.

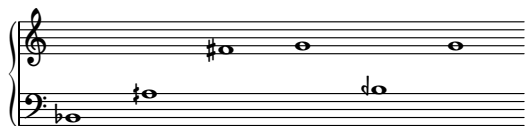
```
>>> numbered_pitch_segment = pitchtools.PitchSegment(  
...     tokens=[-2, -1.5, 6, 7, -1.5, 7],  
...     item_class=pitchtools.NumberedPitch,  
... )  
>>> numbered_pitch_segment  
PitchSegment([-2, -1.5, 6, 7, -1.5, 7])
```

```
>>> show(numbered_pitch_segment)
```



```
>>> named_pitch_segment = pitchtools.PitchSegment(  
...     ['bf', 'aq', 'fs', 'g', 'bqf', 'g'],  
...     item_class=NamedPitch,  
... )  
>>> named_pitch_segment  
PitchSegment(['bf', 'aq', 'fs', 'g', 'bqf', 'g'])
```

```
>>> show(named_pitch_segment)
```



Bases

- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`PitchSegment.has_duplicates`

True if pitch segment has duplicate items. Otherwise false.

```
>>> 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`

Inflection point count of pitch segment.

Returns nonnegative integer.

`(TypedCollection).item_class`

Item class to coerce tokens into.

`PitchSegment.local_maxima`
 Local maxima of pitch segment.
 Returns tuple.

`PitchSegment.local_minima`
 Local minima of pitch segment.
 Returns tuple.

Read/write properties

`(TypedCollection).custom_identifier`
 Gets and sets custom identifier of typed collection.
 Returns string or none.

Methods

`(TypedTuple).count(token)`
 Changes *token* to item.
 Returns count in collection.

`(TypedTuple).index(token)`
 Changes *token* to item.
 Returns index in collection.

`PitchSegment.invert(axis)`
 Inverts pitch segment about *axis*.
 Returns new pitch segment.

`PitchSegment.is_equivalent_under_transposition(expr)`
 True if pitch segment is equivalent to *expr* under transposition. Otherwise false.
 Returns boolean.

`PitchSegment.make_notes(n=None, written_duration=None)`
 Makes first *n* notes in pitch 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)
```



Allows nonassignable *written_duration*:

```
>>> notes = named_pitch_segment.make_notes(4, Duration(5, 16))
>>> staff = Staff(notes)
>>> time_signature = TimeSignature((5, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```

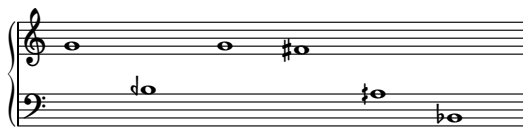


Returns list of notes.

`PitchSegment`.**retrograde**()
Retrograde of pitch segment.

```
>>> result = named_pitch_segment.retrograde()
>>> result
PitchSegment(["g'", 'bqf', "g'", "fs'", 'aqs', 'bf,'])
```

```
>>> show(result)
```

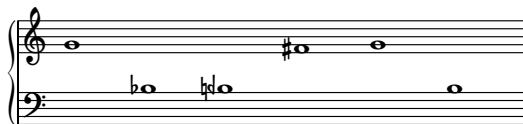


Returns new pitch segment.

`PitchSegment`.**rotate**(*n*)
Rotates pitch segment.

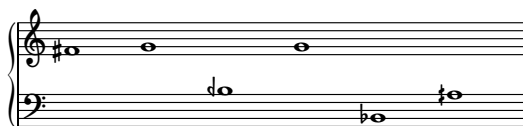
```
>>> result = numbered_pitch_segment.rotate(1)
>>> result
PitchSegment([7, -2, -1.5, 6, 7, -1.5])
```

```
>>> show(result)
```



```
>>> result = named_pitch_segment.rotate(-2)
>>> result
PitchSegment(["fs'", "g'", 'bqf', "g'", 'bf,', 'aqs'])
```

```
>>> show(result)
```



Returns new pitch segment.

`PitchSegment`.**transpose**(*expr*)
Transposes pitch segment by *expr*.

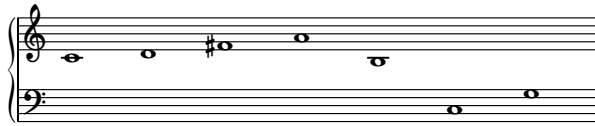
Returns new pitch segment.

Class methods

`PitchSegment`.**from_selection**(*selection*, *item_class=None*, *custom_identifier=None*)
Makes pitch segment from *selection*.

```
>>> staff_1 = Staff("c'4 <d' fs' a'>4 b2")
>>> staff_2 = Staff("c4. r8 g2")
>>> selection = select((staff_1, staff_2))
>>> pitch_segment = pitchtools.PitchSegment.from_selection(
...     selection)
>>> pitch_segment
PitchSegment(["c'", "d'", "fs'", "a'", 'b', 'c', 'g'])
```

```
>>> show(pitch_segment)
```



Returns pitch segment.

Special methods

(TypedTuple) `.__add__ (expr)`

Adds typed tuple to *expr*.

Returns new typed tuple.

(TypedTuple) `.__contains__ (token)`

Change *token* to item and return true if item exists in collection.

Returns none.

(TypedCollection) `.__eq__ (expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) `.__format__ (format_specification='')`

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedTuple) `.__getitem__ (i)`

Gets *i* from type tuple.

Returns item.

(TypedTuple) `.__getslice__ (start, stop)`

Gets slice from *start* to *stop* in typed tuple.

Returns new typed tuple.

(TypedTuple) `.__hash__ ()`

Hashes typed tuple.

Returns integer.

PitchSegment `.__illustrate__ ()`

Illustrates pitch segment.

```
>>> named_pitch_segment = pitchtools.PitchSegment(
...     ['bf', 'aqs', "fs'", "g'", 'bqf', "g'"],
...     item_class=NamedPitch,
... )
>>> show(named_pitch_segment)
```



Returns LilyPond file.

(TypedCollection) `.__iter__ ()`

Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedCollection) .**__makenew__** (*tokens=None, item_class=None, custom_identifier=None*)

Makes new typed collection with optional new values.

Returns new typed collection.

(TypedTuple) .**__mul__** (*expr*)

Multiplies typed tuple by *expr*.

Returns new typed tuple.

(TypedCollection) .**__ne__** (*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedTuple) .**__rmul__** (*expr*)

Multiplies *expr* by typed tuple.

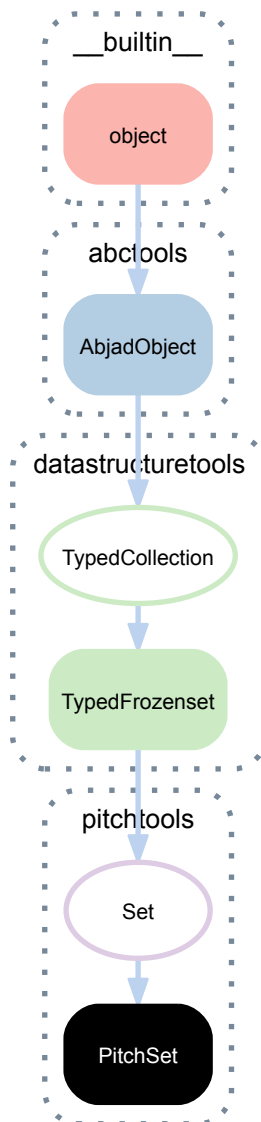
Returns new typed tuple.

(Segment) .**__str__**()

String representation of segment.

Returns string.

13.2.29 pitchtools.PitchSet



class `pitchtools.PitchSet` (*tokens=None, item_class=None, custom_identifer=None*)
 A pitch set.

```
>>> 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=NamedPitch,
... )
>>> named_pitch_set
PitchSet(['bf,', 'aqs', 'bqf', "fs'", "g'])
```

Bases

- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`

- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`PitchSet.duplicate_pitch_classes`

Duplicate pitch-classes in pitch set.

Returns pitch-class set.

`PitchSet.is_pitch_class_unique`

True when pitch set is pitch-class-unique. Otherwise false.

Returns boolean.

`(TypedCollection).item_class`

Item class to coerce tokens into.

Read/write properties

`(TypedCollection).custom_identifier`

Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

`(TypedFrozenSet).copy()`

Copies typed frozen set.

Returns new typed frozen set.

`(TypedFrozenSet).difference(expr)`

Typed frozen set set-minus *expr*.

Returns new typed frozen set.

`(TypedFrozenSet).intersection(expr)`

Set-theoretic intersection of typed frozen set and *expr*.

Returns new typed frozen set.

`PitchSet.invert(axis)`

Inverts pitch set about *axis*.

Returns new pitch set.

`PitchSet.is_equivalent_under_transposition(expr)`

True if pitch set is equivalent to *expr* under transposition. Otherwise false.

Returns boolean.

`(TypedFrozenSet).isdisjoint(expr)`

True when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).issubset(expr)`

True when typed frozen set is a subset of *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).issuperset(expr)`
 True when typed frozen set is a superset of *expr*. Otherwise false.
 Returns boolean.

`(TypedFrozenSet).symmetric_difference(expr)`
 Symmetric difference of typed frozen set and *expr*.
 Returns new typed frozen set.

`PitchSet.transpose(expr)`
 Transposes all pitches in pitch set by *expr*.
 Returns new pitch set.

`(TypedFrozenSet).union(expr)`
 Union of typed frozen set and *expr*.
 Returns new typed frozen set.

Class methods

`PitchSet.from_selection(selection, item_class=None, custom_identifier=None)`
 Makes pitch set from *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)`
 Logical AND of typed frozen set and *expr*.
 Returns new typed frozen set.

`(TypedCollection).__contains__(token)`
 True when typed collection container *token*. Otherwise false.
 Returns boolean.

`(TypedCollection).__eq__(expr)`
 True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.
 Returns boolean.

`(TypedCollection).__format__(format_specification=')`
 Formats typed collection.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`(TypedFrozenSet).__ge__(expr)`
 True when typed frozen set is greater than or equal to *expr*. Otherwise false.
 Returns boolean.

`(TypedFrozenSet).__gt__(expr)`
 True when typed frozen set is greater than *expr*. Otherwise false.
 Returns boolean.

(TypedFrozenSet) .**__hash__**()
Hashes typed frozen set.
Returns integer.

(TypedCollection) .**__iter__**()
Iterates typed collection.
Returns generator.

(TypedFrozenSet) .**__le__**(*expr*)
True when typed frozen set is less than or equal to *expr*. Otherwise false.
Returns boolean.

(TypedCollection) .**__len__**()
Length of typed collection.
Returns nonnegative integer.

(TypedFrozenSet) .**__lt__**(*expr*)
True when typed frozen set is less than *expr*. Otherwise false.
Returns boolean.

(TypedCollection) .**__makenew__**(*tokens=None, item_class=None, custom_identifier=None*)
Makes new typed collection with optional new values.
Returns new typed collection.

(TypedFrozenSet) .**__ne__**(*expr*)
True when typed frozen set is not equal to *expr*. Otherwise false.
Returns boolean.

(TypedFrozenSet) .**__or__**(*expr*)
Logical OR of typed frozen set and *expr*.
Returns new typed frozen set.

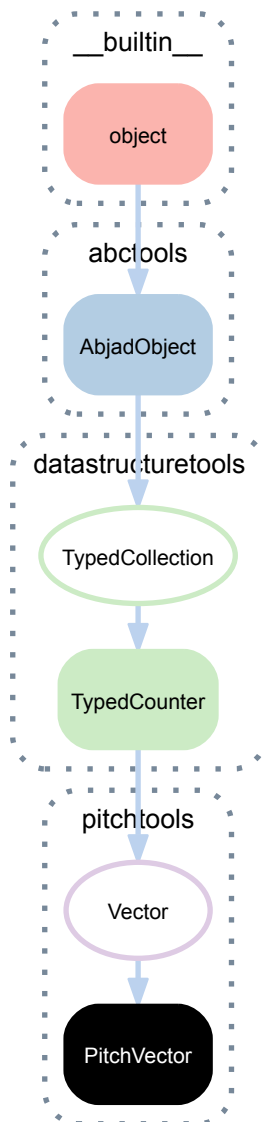
(AbjadObject) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(Set) .**__str__**()
String representation of set.
Returns string.

(TypedFrozenSet) .**__sub__**(*expr*)
Subtracts *expr* from typed frozen set.
Returns new typed frozen set.

(TypedFrozenSet) .**__xor__**(*expr*)
Logical XOR of typed frozen set and *expr*.
Returns new typed frozen set.

13.2.30 pitchtools.PitchVector



class `pitchtools.PitchVector` (*tokens=None, item_class=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
Gets and sets custom identifier of typed collection.
Returns string or none.

Methods

`(TypedCounter).clear()`
Clears typed counter.
Returns none.

`(TypedCounter).copy()`
Copies typed counter.
Returns new typed counter.

`(TypedCounter).elements()`
Elements in typed counter.

`(TypedCounter).items()`
Items in typed counter.
Returns tuple.

`(TypedCounter).iteritems()`
Iterates items in typed counter.
Yields items.

`(TypedCounter).iterkeys()`
Iterates keys in typed counter.

`(TypedCounter).intervalues()`
Iterates values in typed counter.

`(TypedCounter).keys()`
Keys in typed counter.

`(TypedCounter).most_common(n=None)`
Please document.

`(TypedCounter).subtract(iterable=None, **kwargs)`
Stracts *iterable* from typed counter.

`(TypedCounter).update(iterable=None, **kwargs)`
Updates typed counter with *iterable*.

`(TypedCounter).values()`
Values of typed counter.

`(TypedCounter).viewitems()`
Please document.

`(TypedCounter).viewkeys()`
Please document.

`(TypedCounter).viewvalues()`
Please document.

Class methods

`PitchVector.from_selection(selection, item_class=None, custom_identifier=None)`
Makes pitch vector from *selection*.

Returns pitch vector.

Special methods

`(TypedCounter).__add__(expr)`

Adds typed counter to *expr*.

Returns new typed counter.

`(TypedCounter).__and__(expr)`

Logical AND of typed counter and *expr*.

Returns new typed counter.

`(TypedCollection).__contains__(token)`

True when typed collection container *token*. Otherwise false.

Returns boolean.

`(TypedCounter).__delitem__(token)`

Deletes *token* from typed counter.

Returns none.

`(TypedCollection).__eq__(expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`

Formats typed collection.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(TypedCounter).__getitem__(token)`

Gets *token* from typed counter.

Returns item.

`(TypedCollection).__iter__()`

Iterates typed collection.

Returns generator.

`(TypedCollection).__len__()`

Length of typed collection.

Returns nonnegative integer.

`(TypedCollection).__makenew__(tokens=None, item_class=None, custom_identifier=None)`

Makes new typed collection with optional new values.

Returns new typed collection.

`(TypedCounter).__missing__(token)`

Returns zero.

Returns zero.

`(TypedCollection).__ne__(expr)`

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(TypedCounter) .**__or__** (*expr*)
Logical OR of typed counter and *expr*.
Returns new typed counter.

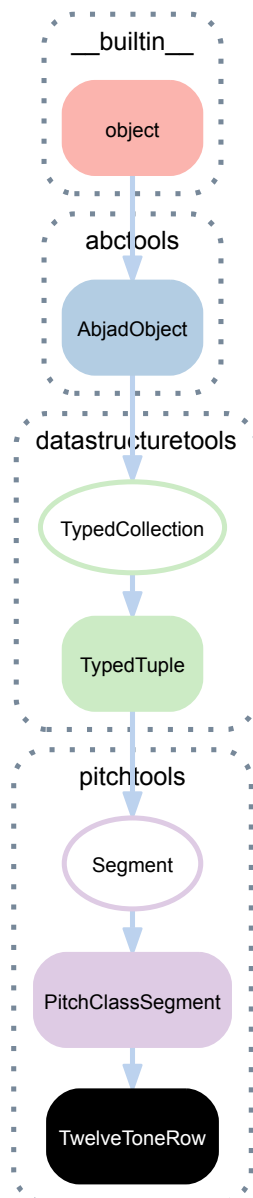
(AbjadObject) .**__repr__** ()
Gets interpreter representation of Abjad object.
Returns string.

(TypedCounter) .**__setitem__** (*token*, *value*)
Sets typed counter *token* to *value*.
Returns none.

(Vector) .**__str__** ()
String representation of vector.
Returns string.

(TypedCounter) .**__sub__** (*expr*)
Subtracts *expr* from typed counter.
Returns new typed counter.

13.2.31 pitchtools.TwelveToneRow



class `pitchtools.TwelveToneRow` (*tokens=None, custom_identifier=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.

Read/write properties

`(TypedCollection).custom_identifier`
 Gets and sets custom identifier of typed collection.
 Returns string or none.

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])
```

Returns new pitch-class segment.

`(TypedTuple).count(token)`
 Changes *token* to item.

Returns count in collection.

`(TypedTuple).index(token)`
 Changes *token* to item.

Returns 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])
```

Returns new pitch-class segment.

(PitchClassSegment) **.is_equivalent_under_transposition** (*expr*)
 True if equivalent under transposition to *expr*. Otherwise False.

Returns boolean.

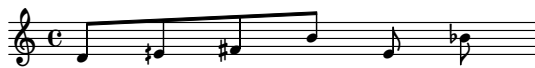
(PitchClassSegment) **.make_notes** (*n=None*, *written_duration=None*)
 Make first *n* notes in pitch class segment.

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 = TimeSignature((5, 4))
>>> attach(time_signature, 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])
```

Returns new pitch-class segment.

(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])
```

Returns 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'])
```

Returns 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])
```

Returns new pitch-class segment.

Class methods

TwelveToneRow **.from_selection** (*selection*, *item_class=None*, *custom_identifier=None*)

Makes twelve-tone row from *selection*.

Returns twelve-tone row.

Special methods

(TypedTuple) **.__add__** (*expr*)

Adds typed tuple to *expr*.

Returns new typed tuple.

(TypedTuple) **.__contains__** (*token*)

Change *token* to item and return true if item exists in collection.

Returns none.

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) **.__format__** (*format_specification=''*)

Formats typed collection.

Set *format_specification* to ' or 'storage'. Interprets ' equal to 'storage'.

Returns string.

(TypedTuple) **.__getitem__** (*i*)

Gets *i* from type tuple.

Returns item.

TwelveToneRow **.__getslice__** (*start*, *stop*)

Gets items from *start* to *stop* in twelve-tone row.

Returns pitch-class segment.

(TypedTuple) **.__hash__** ()

Hashes typed tuple.

Returns integer.

`(TypedCollection).__iter__()`
 Iterates typed collection.
 Returns generator.

`(TypedCollection).__len__()`
 Length of typed collection.
 Returns nonnegative integer.

`TwelveToneRow.__makenew__(tokens=None, custom_identifier=None)`
 Makes new twelve-tone row with optional *tokens* and *custom_identifier*.
 Returns new twelve-tone row.

`TwelveToneRow.__mul__(expr)`
 Multiplies twelve-tone row by *expr*.
 Returns pitch-class segment.

`(TypedCollection).__ne__(expr)`
 True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.
 Returns boolean.

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

`TwelveToneRow.__rmul__(expr)`
 Multiplies *expr* by twelve-tone row.
 Returns pitch-class segment.

`(Segment).__str__()`
 String representation of segment.
 Returns string.

13.3 Functions

13.3.1 `pitchtools.apply_accidental_to_named_pitch`

`pitchtools.apply_accidental_to_named_pitch(named_pitch, accidental=None)`
 Apply *accidental* to *named_pitch*:

```
>>> pitch = NamedPitch("cs' ")
>>> pitchtools.apply_accidental_to_named_pitch(pitch, 'f')
NamedPitch("c' ")
```

Returns new named pitch.

13.3.2 `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 = Clef('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.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.

13.3.3 `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.

13.3.4 `pitchtools.get_named_pitch_from_pitch_carrier`

`pitchtools.get_named_pitch_from_pitch_carrier` (*pitch_carrier*)

Gets named pitch from *pitch_carrier*.

```
>>> pitch = 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 value error when *pitch_carrier* carries no pitch.

Raises value error when *pitch_carrier* carries more than one pitch.

Returns named pitch.

13.3.5 `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.

13.3.6 `pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list`

`pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list` (*notes*,
sub-
run_tokens)

Insert and transpose nested subruns in *pitch_class_number_list* according to *subrun_tokens*:

```
>>> notes = [Note(p, (1, 4)) for p in [0, 2, 7, 9, 5, 11, 4]]
>>> subrun_tokens = [(0, [2, 4]), (4, [3, 1])]
>>> pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list(
...     notes, subrun_tokens)

>>> 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_tokens* to a list of zero or more (*index*, *length_list*) pairs.

For each (*index*, *length_list*) pair in *subrun_tokens* 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.

13.3.7 `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.

13.3.8 `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.

13.3.9 `pitchtools.iterate_named_pitch_pairs_in_expr`

`pitchtools.iterate_named_pitch_pairs_in_expr(expr)`

Iterates 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))
>>> clef = Clef('bass')
>>> attach(clef, score[1])
>>> show(score)
```



```
>>> 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.

13.3.10 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 = spannertools.Beam()
>>> attach(beam, staff[:])
```

```
>>> for x in pitchtools.list_named_pitches_in_expr(beam):
...     x
...
NamedPitch("c'")
NamedPitch("d'")
NamedPitch("e'")
NamedPitch("f'")
```

Returns tuple.

13.3.11 `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.

13.3.12 `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.

13.3.13 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.

13.3.14 `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.

13.3.15 `pitchtools.list_pitch_numbers_in_expr`

`pitchtools.list_pitch_numbers_in_expr` (*expr*)

List pitch numbers in *expr*:

```
>>> tuplet = scoretools.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.

13.3.16 `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.

13.3.17 `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.

13.3.18 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 = Clef('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.

13.3.19 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.

13.3.20 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 = scoretools.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.

13.3.21 `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.

13.3.22 `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.

13.3.23 `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 = scoretools.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.

13.3.24 `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.

13.3.25 `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.

13.3.26 `pitchtools.suggest_clef_for_named_pitches`

`pitchtools.suggest_clef_for_named_pitches` (*pitches*)

Suggest clef for named *pitches*:

```
>>> staff = Staff(scoretools.make_notes(range(-12, -6), [(1, 4)]))
>>> pitchtools.suggest_clef_for_named_pitches(staff)
Clef('bass')
```

Suggest clef based on minimal number of ledger lines.

Returns clef.

13.3.27 `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 = NamedPitch(0)

>>> pitchtools.transpose_named_pitch_by_numbered_interval_and_respell(
...     pitch, 1, 0.5)
NamedPitch("dtqf' ")
```

Returns new named pitch.

13.3.28 `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*.

13.3.29 `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.

13.3.30 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.

13.3.31 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*:

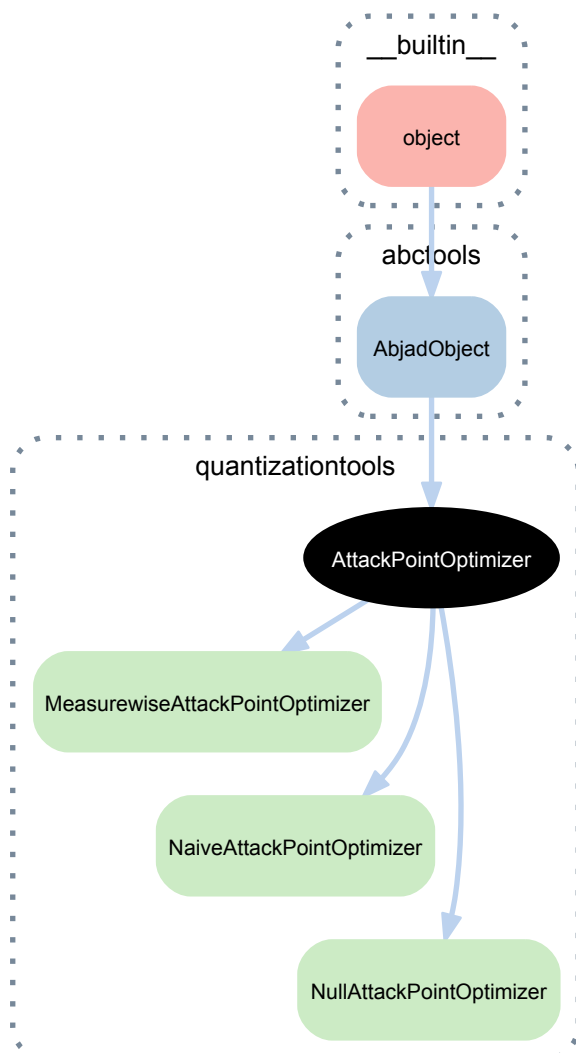
```
>>> pitch_numbers = [-30, -18, -6, 6, 18, 30, 42]
>>> for n in pitch_numbers:
...     n, pitchtools.transpose_pitch_number_by_octave_transposition_mapping(
...         n, mapping)
...
(-30, 6)
(-18, 6)
(-6, 18)
(6, 18)
(18, 18)
(30, 30)
(42, 30)
```

And so on.

Returns pitch number.

14.1 Abstract classes

14.1.1 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 logical tie, but may not alter the overall duration of that logical tie.

They effectively “clean up” notation, post-quantization.

Bases

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

Special methods

`AttackPointOptimizer.__call__(expr)`
Calls attack-point optimizer.

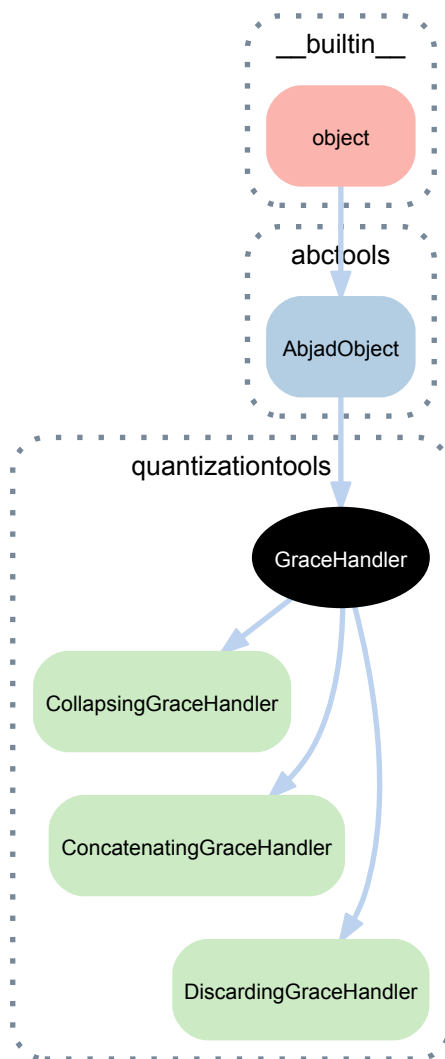
`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

14.1.2 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)`

Calls grace handler.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

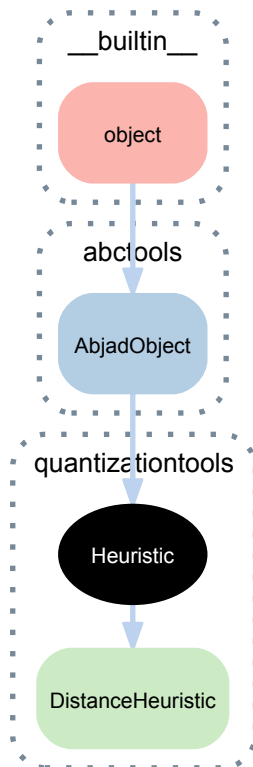
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.1.3 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)`

Calls heuristic.

Returns none.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

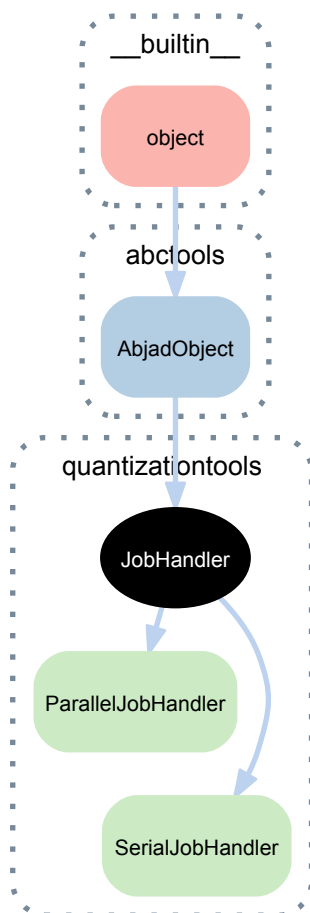
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.1.4 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)`

Calls job handler.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `''` or `'storage'`. Interprets `''` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

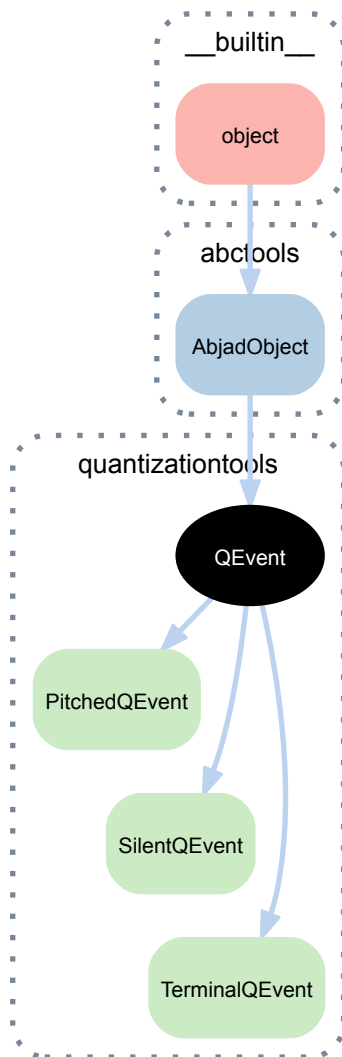
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.1.5 quantizationtools.QEvent



class `quantizationtools.QEvent` (*offset=0, 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.

Special methods

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

QEvent.**__lt__**(*expr*)

True when *expr* is a q-event with offset greater than that of this q-event. Otherwise false.

Returns boolean.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

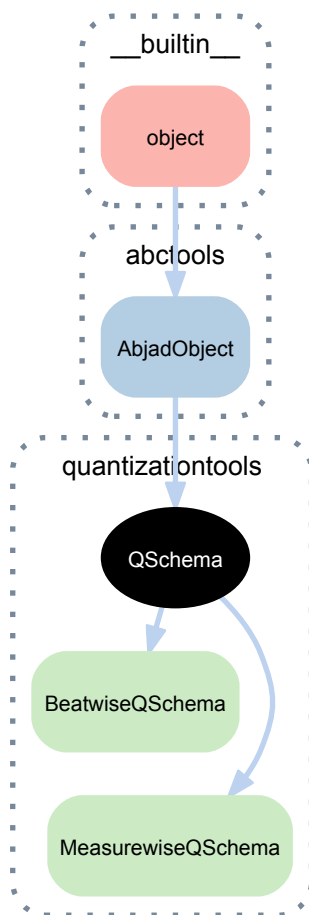
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.1.6 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 indicators 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)`

Calls `QSchema` on *duration*.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`QSchema.__format__(format_specification='')`

Formats q-event.

Set *format_specification* to ‘’ or ‘*storage*’. Interprets ‘’ equal to ‘*storage*’.

Returns string.

`QSchema.__getitem__(i)`

Gets item *i* in `QSchema`.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

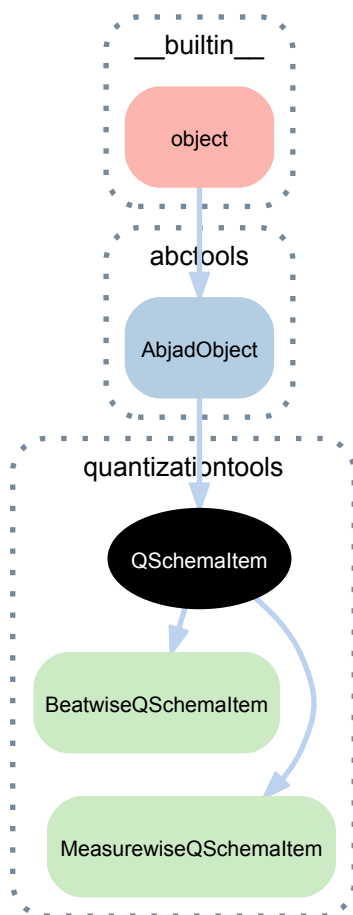
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.1.7 quantizationtools.QSchemaItem



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.tempo`

The optionally defined tempo.

Returns tempo or none.

Special methods

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

QSchemaItem.**__format__**(*format_specification*='')

Formats q schema item.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

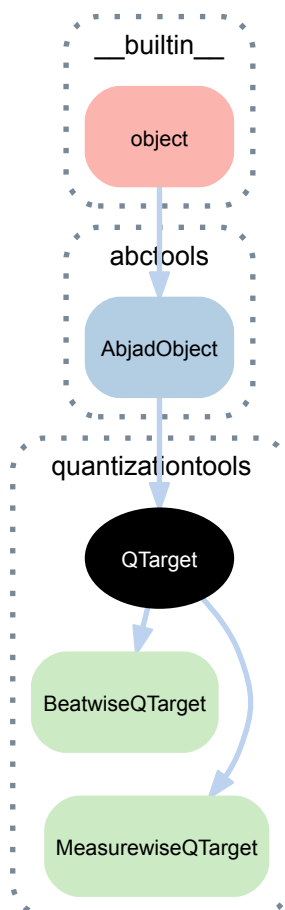
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.1.8 quantizationtools.QTarget



class quantizationtools.**QTarget** (*items=None*)

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`

Beats of q-target.

`QTarget.duration_in_ms`

Duration of q-target in milliseconds.

Returns duration.

`QTarget.item_class`

Item class of q-target.

`QTarget.items`

Items of q-target.

Special methods

`QTarget.__call__`(*q_event_sequence*, *grace_handler=None*, *heuristic=None*, *job_handler=None*,
attack_point_optimizer=None, *attach_tempos=True*)

Calls q-target.

(`AbjadObject`).`__eq__`(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(`AbjadObject`).`__format__`(*format_specification=''*)

Formats object.

Set *format_specification* to `''` or `'storage'`. Interprets `''` equal to `'storage'`.

Returns string.

(`AbjadObject`).`__ne__`(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

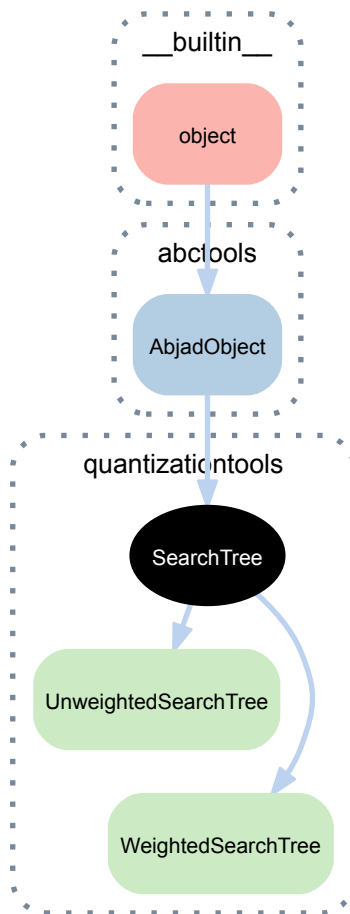
Returns boolean.

(`AbjadObject`).`__repr__`()

Gets interpreter representation of Abjad object.

Returns string.

14.1.9 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)`

Calls search tree.

`SearchTree.__eq__(expr)`

True when *expr* is a search tree with definition equal to that of this search tree. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

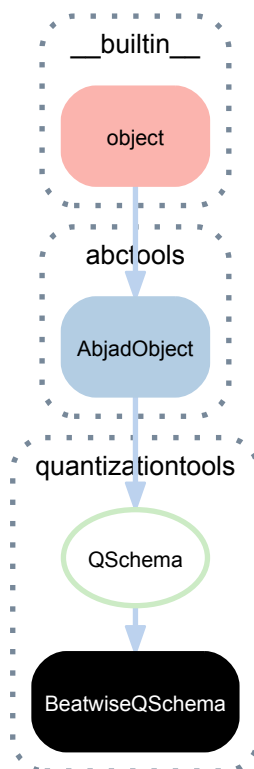
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2 Concrete classes

14.2.1 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:

```
>>> print format(q_schema, 'storage')
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=indicatortools.Tempo(
    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 = Tempo((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: 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: 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)
```

```
>>> print format(q_schema[0]['search_tree'])
quantizationtools.UnweightedSearchTree(
  definition={
    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,
},
)

```

```

>>> print format(q_schema[1]['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,
},
)

```

```

>>> 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`
 Default beatspan of beatwise q-schema.

`BeatwiseQSchema.item_class`
 The schema's item class.
 Returns beatwise q-schema item.

`(QSchema).items`
 The item dictionary.

`(QSchema).search_tree`
 The default search tree.

`BeatwiseQSchema.target_class`
 Target class of beatwise q-schema.
 Returns beatwise q-target.

`BeatwiseQSchema.target_item_class`
 Target item class of beatwise q-schema.
 Returns q-target beat.

`(QSchema).tempo`
 The default tempo.

Special methods

`(QSchema).__call__(duration)`
 Calls `QSchema` on *duration*.

`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

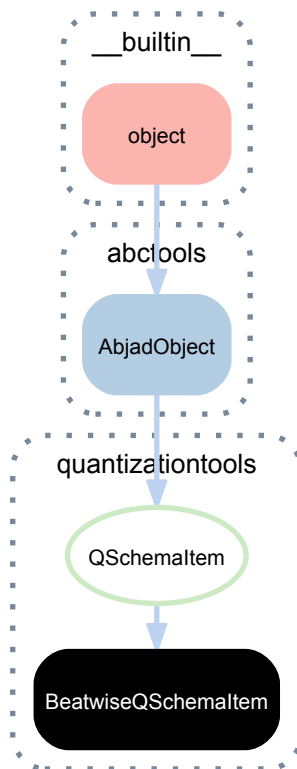
`(QSchema).__format__(format_specification='')`
 Formats q-event.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`(QSchema).__getitem__(i)`
 Gets item *i* in `QSchema`.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

14.2.2 quantizationtools.BeatwiseQSchemaItem



class `quantizationtools.BeatwiseQSchemaItem` (*beatspan=None*, *search_tree=None*, *tempo=None*)
BeatwiseQSchemaItem represents a change of state in the timeline of an unmetered quantization process.

```
>>> q_schema_item = quantizationtools.BeatwiseQSchemaItem()
>>> print format(q_schema_item)
quantizationtools.BeatwiseQSchemaItem()
```

Define a change in tempo:

```
>>> q_schema_item = quantizationtools.BeatwiseQSchemaItem(
...     tempo=((1, 4), 60),
... )
>>> print format(q_schema_item)
quantizationtools.BeatwiseQSchemaItem(
    tempo=indicatortools.Tempo(
        durationtools.Duration(1, 4),
        60
    ),
)
```

Define a change in beatspan:

```
>>> q_schema_item = quantizationtools.BeatwiseQSchemaItem(
...     beatspan=(1, 8),
... )
>>> print format(q_schema_item)
quantizationtools.BeatwiseQSchemaItem(
    beatspan=durationtools.Duration(1, 8),
)
```

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).tempo`

The optionally defined tempo.

Returns tempo or none.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(QSchemaItem).__format__(format_specification='')`

Formats q schema item.

Set *format_specification* to `''` or `'storage'`. Interprets `''` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

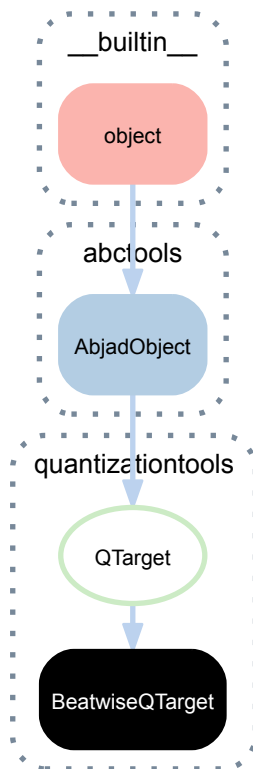
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.3 quantizationtools.BeatwiseQTarget



class `quantizationtools.BeatwiseQTarget` (*items=None*)

A beat-wise quantization target.

Not composer-safe.

Used internally by `Quantizer`.

Bases

- `quantizationtools.QTarget`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`BeatwiseQTarget.beats`

Beats of beatwise q-target.

`(QTarget).duration_in_ms`

Duration of q-target in milliseconds.

Returns duration.

`BeatwiseQTarget.item_class`

Item class of beatwise q-target.

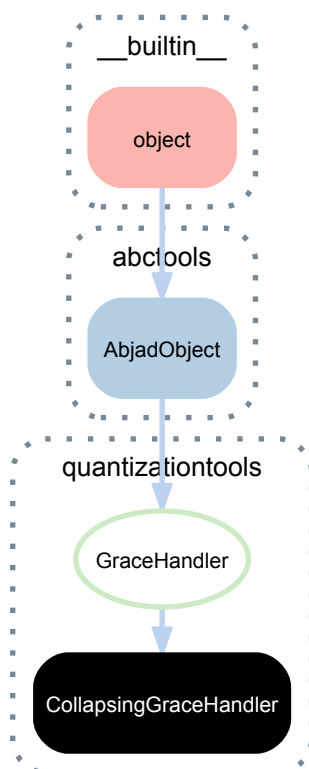
`(QTarget).items`

Items of q-target.

Special methods

- (QTarget) .**__call__**(*q_event_sequence*, *grace_handler=None*, *heuristic=None*,
job_handler=None, *attack_point_optimizer=None*, *attach_tempos=True*)
 Calls q-target.
- (AbjadObject) .**__eq__**(*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.
- (AbjadObject) .**__format__**(*format_specification=''*)
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.
- (AbjadObject) .**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.
- (AbjadObject) .**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

14.2.4 quantizationtools.CollapsingGraceHandler



class quantizationtools.CollapsingGraceHandler

A GraceHandler which collapses pitch information into a single chord rather than creating a grace container.

Bases

- `quantizationtools.GraceHandler`

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

Special methods

`CollapsingGraceHandler.__call__(q_events)`

Calls collapsing grace handler.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

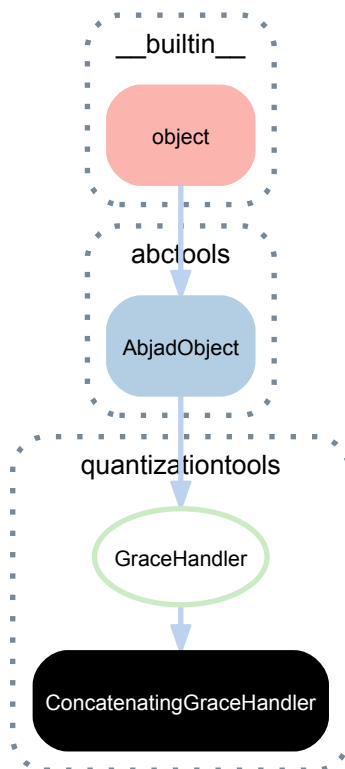
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.5 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.

Bases

- `quantizationtools.GraceHandler`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ConcatenatingGraceHandler.grace_duration`

Grace duration of concatenating grace handler.

Returns duration.

Special methods

`ConcatenatingGraceHandler.__call__(q_events)`

Calls concatenating grace handler.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

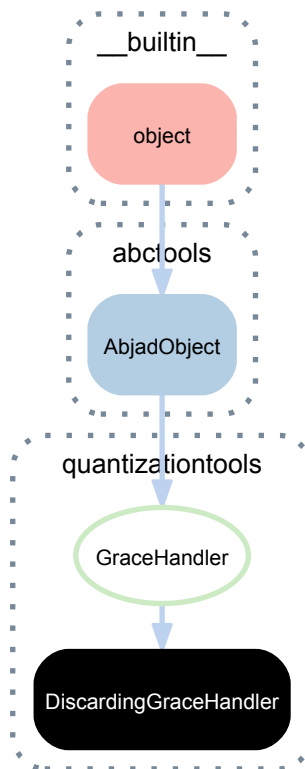
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.6 quantizationtools.DiscardingGraceHandler



class `quantizationtools.DiscardingGraceHandler`

Concrete `GraceHandler` subclass which discards all but final `QEvent` attached to an offset.

Does not create `GraceContainers`.

Bases

- `quantizationtools.GraceHandler`
- `abctools.AbjadObject`
- `__builtin__.object`

Special methods

`DiscardingGraceHandler.__call__(q_events)`

Calls `idscarind` grace handler.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

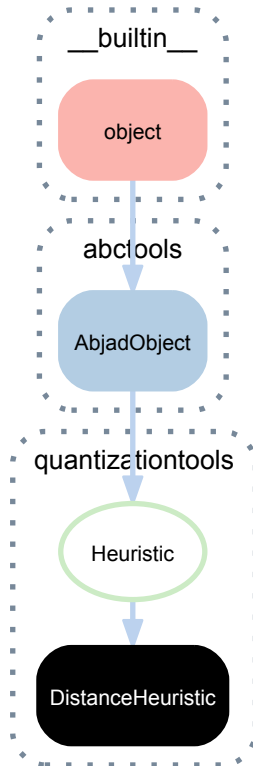
`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

14.2.7 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)`
 Calls heuristic.
 Returns none.

`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

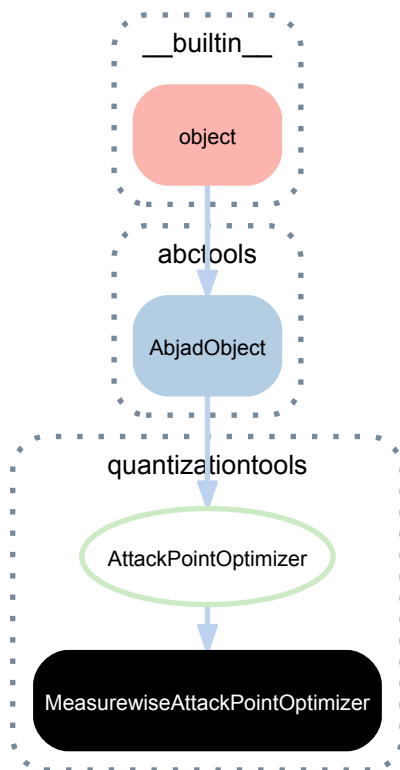
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.8 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.

Bases

- quantizationtools.AttackPointOptimizer
- abctools.AbjadObject
- __builtin__.object

Special methods

`MeasurewiseAttackPointOptimizer.__call__(expr)`

Calls measurewise attack-point optimizer.

Returns none.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

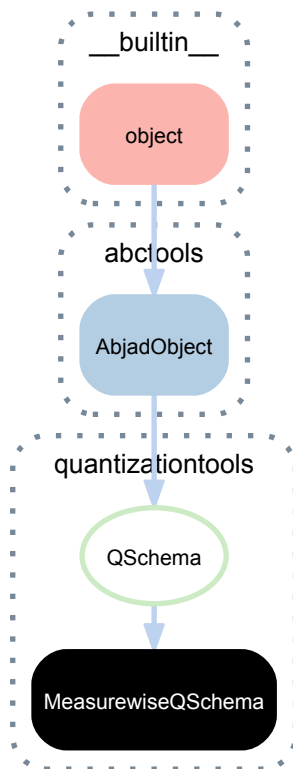
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.9 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:

```

>>> print format(q_schema, 'storage')
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=indicatortools.Tempo(
    durationtools.Duration(1, 4),
    60
  ),
  time_signature=indicatortools.TimeSignature(
    (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 = TimeSignature((3, 4))
>>> tempo = Tempo((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: 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: 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': TimeSignature((7, 32))}
>>> b = {'time_signature': TimeSignature((3, 4))}
>>> c = {'time_signature': TimeSignature((5, 8))}
```

```
>>> settings = {
...     2: a,
...     4: b,
...     6: c,
... }
```

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(settings)
```

```
>>> q_schema[0]['time_signature']
TimeSignature((4, 4))
```

```
>>> q_schema[1]['time_signature']
TimeSignature((4, 4))
```

```
>>> q_schema[2]['time_signature']
TimeSignature((7, 32))
```

```
>>> q_schema[3]['time_signature']
TimeSignature((7, 32))
```

```
>>> q_schema[4]['time_signature']
TimeSignature((3, 4))
```

```
>>> q_schema[5]['time_signature']
TimeSignature((3, 4))
```

```
>>> q_schema[6]['time_signature']
TimeSignature((5, 8))
```

```
>>> q_schema[1000]['time_signature']
TimeSignature((5, 8))
```

The following is equivalent to the above schema definition:

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(
...     (2, {'time_signature': TimeSignature((7, 32))}),
...     (4, {'time_signature': TimeSignature((3, 4))}),
...     (6, {'time_signature': TimeSignature((5, 8))}),
... )
```

Return MeasurewiseQSchema instance.

Bases

- `quantizationtools.QSchema`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`MeasurewiseQSchema.item_class`

Item class of measurewise q-schema.

Returns MeasurewiseQSchemaItem.

`(QSchema).items`

The item dictionary.

`(QSchema).search_tree`

The default search tree.

`MeasurewiseQSchema.target_class`

Target class of measurewise q-schema.

Returns MeasurewiseQTarget.

`MeasurewiseQSchema.target_item_class`

Target item class of measurewise q-schema.

Returns QTargetMeasure.

`(QSchema).tempo`

The default tempo.

`MeasurewiseQSchema.time_signature`

Default time signature of measurewise q-schema.

Returns time signature.

`MeasurewiseQSchema.use_full_measure`

The full-measure-as-beatspan default.

Returns boolean.

Special methods

`(QSchema).__call__(duration)`

Calls `QSchema` on *duration*.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(QSchema).__format__(format_specification='')`

Formats q-event.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(QSchema).__getitem__(i)`

Gets item *i* in `QSchema`.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

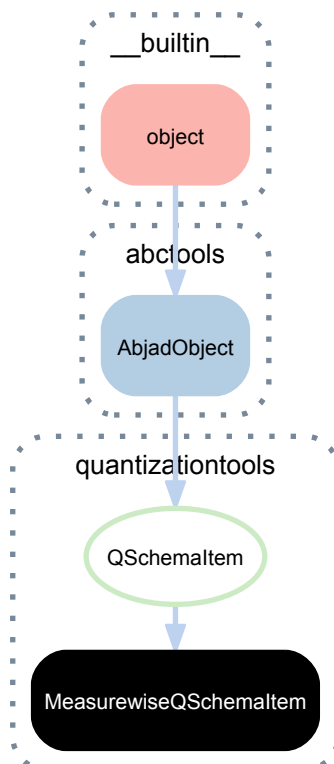
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.10 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 format(q_schema_item)
quantizationtools.MeasurewiseQSchemaItem()
```

Define a change in tempo:

```
>>> q_schema_item = quantizationtools.MeasurewiseQSchemaItem(
...     tempo=((1, 4), 60),
... )
>>> print format(q_schema_item)
quantizationtools.MeasurewiseQSchemaItem(
    tempo=indicatortools.Tempo(
        durationtools.Duration(1, 4),
        60
    ),
)
```

Define a change in time signature:

```
>>> q_schema_item = quantizationtools.MeasurewiseQSchemaItem(
...     time_signature=(6, 8),
... )
>>> print format(q_schema_item)
quantizationtools.MeasurewiseQSchemaItem(
    time_signature=indicatortools.TimeSignature(
        (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*).**tempo**

The optionally defined tempo.

Returns tempo or none.

`MeasurewiseQSchemaItem.time_signature`

The optionally defined TimeSignature.

Returns time signature or none

`MeasurewiseQSchemaItem.use_full_measure`

If True, use the full measure as the beatspan.

Returns boolean or none.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(QSchemaItem).__format__(format_specification='')`

Formats q schema item.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

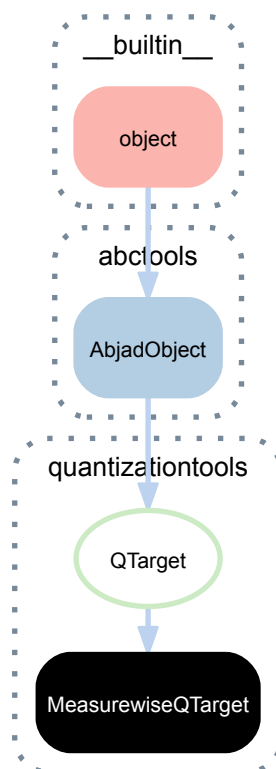
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.11 quantizationtools.MeasurewiseQTarget



class `quantizationtools.MeasurewiseQTarget` (*items=None*)
 A measure-wise quantization target.
 Not composer-safe.
 Used internally by `Quantizer`.

Bases

- `quantizationtools.QTarget`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`MeasurewiseQTarget.beats`
 Beats of measurewise q-target.
 Returns tuple.

`(QTarget).duration_in_ms`
 Duration of q-target in milliseconds.
 Returns duration.

`MeasurewiseQTarget.item_class`
 Item class of measurewise q-target.
 Returns q-target measure class.

`(QTarget).items`
 Items of q-target.

Special methods

`(QTarget).__call__(q_event_sequence, grace_handler=None, heuristic=None, job_handler=None, attack_point_optimizer=None, attach_tempos=True)`
 Calls q-target.

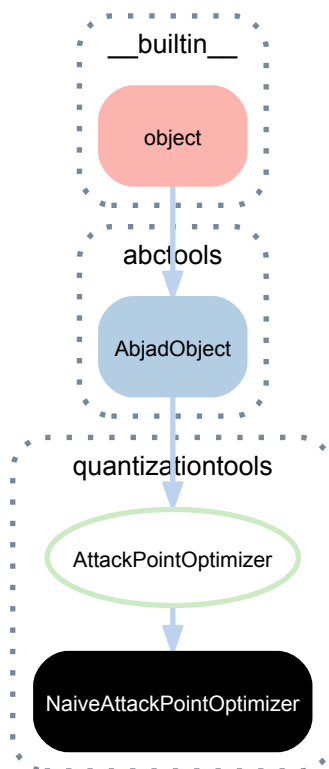
`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

14.2.12 quantizationtools.NaiveAttackPointOptimizer



class `quantizationtools.NaiveAttackPointOptimizer`

Concrete `AttackPointOptimizer` subclass which optimizes attack points by fusing tie leaves within logical ties with leaf durations decreasing monotonically.

TieChains will be partitioned into sub-TieChains if leaves are found with `TempoMarks` attached.

Bases

- `quantizationtools.AttackPointOptimizer`
- `abctools.AbjadObject`
- `__builtin__.object`

Special methods

`NaiveAttackPointOptimizer.__call__(expr)`

Calls naive attack-point optimizer.

Returns none.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

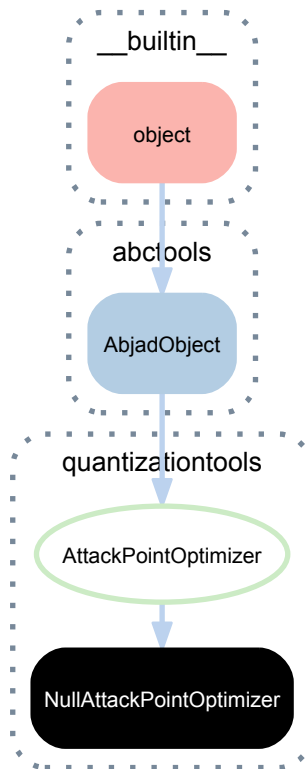
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.13 quantizationtools.NullAttackPointOptimizer



class quantizationtools.**NullAttackPointOptimizer**

Concrete AttackPointOptimizer subclass which performs no attack point optimization.

Bases

- `quantizationtools.AttackPointOptimizer`
- `abctools.AbjadObject`
- `__builtin__.object`

Special methods

NullAttackPointOptimizer.**__call__**(*expr*)

Calls null attack-point optimizer.

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

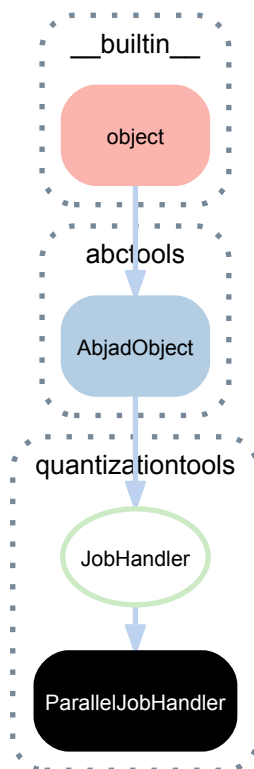
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.14 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*)

Calls parallel job handler.

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

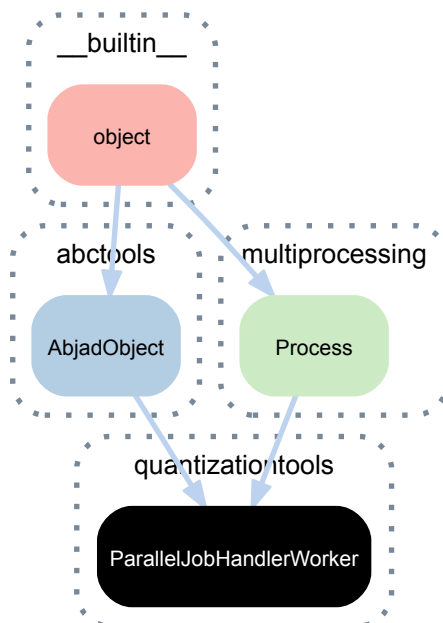
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.15 quantizationtools.ParallelJobHandlerWorker



class quantizationtools.**ParallelJobHandlerWorker**(*job_queue=None*,
sult_queue=None)

re-

Worker process which runs QuantizationJobs.

Not composer-safe.

Used internally by ParallelJobHandler.

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()`
Runs parallel job handler worker.

Returns none.

`(Process) .start()`
Start child process

`(Process) .terminate()`
Terminate process; sends SIGTERM signal or uses `TerminateProcess()`

Special methods

`(AbjadObject) .__eq__ (expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject) .__format__ (format_specification='')`
Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

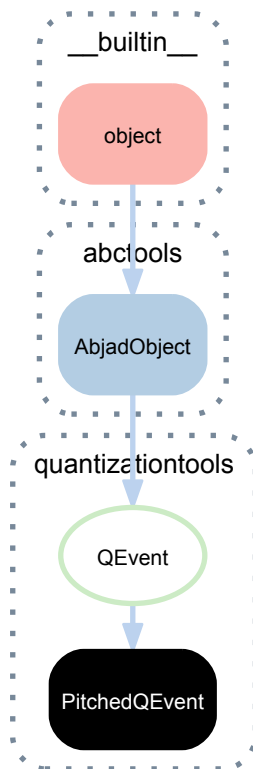
Returns string.

`(AbjadObject) .__ne__ (expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(Process) .__repr__()`

14.2.16 quantizationtools.PitchedQEvent



class `quantizationtools.PitchedQEvent` (*offset=0, pitches=None, 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)
>>> print format(q_event, 'storage')
quantizationtools.PitchedQEvent(
  offset=durationtools.Offset(1000, 1),
  pitches=(
    pitchtools.NamedPitch("c'"),
    pitchtools.NamedPitch("cs'"),
    pitchtools.NamedPitch("e'"),
  ),
)
  
```

Bases

- `quantizationtools.QEvent`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`PitchedQEvent.attachments`
Attachments of pitched q-event.

`(QEvent).index`
The optional index, for sorting `QEvents` with identical offsets.

`(QEvent).offset`
The offset in milliseconds of the event.

`PitchedQEvent.pitches`
 Pitches of pitched q-event.

Special methods

`PitchedQEvent.__eq__(expr)`
 True when *expr* is a pitched q-event with offset, pitches, attachments and index equal to those of this pitched q-event. Otherwise false.
 Returns boolean.

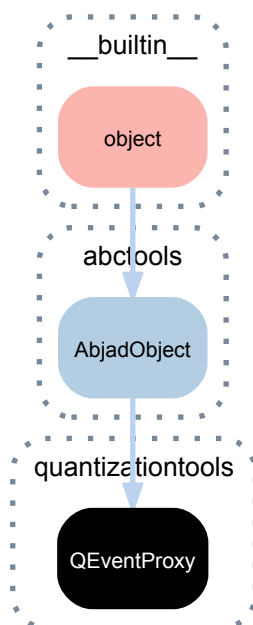
`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`(QEvent).__lt__(expr)`
 True when *expr* is a q-event with offset greater than that of this q-event. Otherwise false.
 Returns boolean.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

14.2.17 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)
>>> print format(proxy, 'storage')
quantizationtools.QEventProxy(

```



```

    q_event=quantizationtools.PitchedQEvent(
        offset=durationtools.Offset(130, 1),
        pitches=(
            pitchtools.NamedPitch("c'"),
            pitchtools.NamedPitch("cs'"),
            pitchtools.NamedPitch("e'"),
        ),
        offset=durationtools.Offset(1, 2),
    )

```

Not composer-safe.

Used internally by Quantizer.

Bases

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

Read-only properties

`QEventProxy.index`
Index of q-event proxy.

`QEventProxy.offset`
Offset of q-event proxy.

`QEventProxy.q_event`
Q-event of q-event proxy.

Special methods

`QEventProxy.__eq__(expr)`
True when *expr* is a q-event proxy with offset and q-event equal to those of this q-event proxy. Otherwise false.

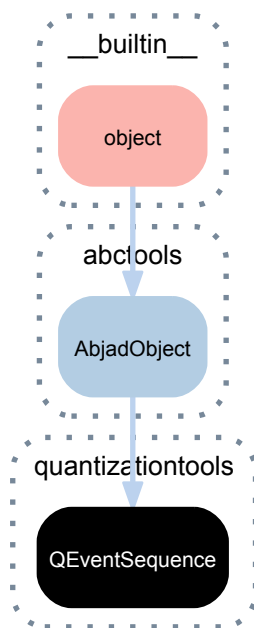
Returns boolean.

`QEventProxy.__format__(format_specification='')`
Formats q-event.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

14.2.18 quantizationtools.QEventSequence



class `quantizationtools.QEventSequence` (*sequence=None*)

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:
...     print format(q_event, 'storage')
...
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(0, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(1000, 1),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(1500, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(2750, 1),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(3250, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.TerminalQEvent(
```

```
offset=durationtools.Offset(4000, 1),
)
```

Bases

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

Read-only properties

`QEventSequence.duration_in_ms`

Duration in milliseconds of the `QEventSequence`:

```
>>> sequence.duration_in_ms
Duration(4000, 1)
```

Return `Duration` instance.

`QEventSequence.sequence`

Sequence of q-events.

```
>>> for q_event in sequence.sequence:
...     print format(q_event, 'storage')
...
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(0, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(1000, 1),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(1500, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(2750, 1),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(3250, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.TerminalQEvent(
    offset=durationtools.Offset(4000, 1),
)
```

Returns tuple.

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:
...     print format(q_event, 'storage')
...
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(0, 1),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(250, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(750, 1),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(1750, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(3000, 1),
)
quantizationtools.TerminalQEvent(
    offset=durationtools.Offset(4000, 1),
)
```

Returns 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:
...     print format(q_event, 'storage')
...
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(0, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(250, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(750, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(1750, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(3000, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
```

```
quantizationtools.TerminalQEvent(
    offset=durationtools.Offset(4000, 1),
)
```

Returns QEventSequence instance.

QEventSequence.**from_millisecond_pitch_pairs** (*pairs*)

Convert millisecond-duration:pitch 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:
...     print format(q_event, 'storage')
...
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(0, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(250, 1),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(750, 1),
    pitches=(
        pitchtools.NamedPitch("d'"),
        pitchtools.NamedPitch("ef'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(1750, 1),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(3000, 1),
    pitches=(
        pitchtools.NamedPitch("cs'"),
    ),
)
quantizationtools.TerminalQEvent(
    offset=durationtools.Offset(4000, 1),
)
```

Returns QEventSequence instance.

QEventSequence.**from_tempo_scaled_durations** (*durations*, *tempo=None*)

Convert durations, scaled by tempo into a QEventSequence:

```
>>> tempo = Tempo((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:
...     print format(q_event, 'storage')
...
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(0, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.SilentQEvent(
```

```
        offset=durationtools.Offset(10000, 29),
    )
    quantizationtools.PitchedQEvent(
        offset=durationtools.Offset(17500, 29),
        pitches=(
            pitchtools.NamedPitch("c'"),
        ),
    )
    quantizationtools.SilentQEvent(
        offset=durationtools.Offset(20000, 29),
    )
    quantizationtools.TerminalQEvent(
        offset=durationtools.Offset(40000, 29),
    )
```

Returns 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 gqs'2")
>>> tempo = Tempo((1, 4), 72)
```

```
>>> sequence = \
...     quantizationtools.QEventSequence.from_tempo_scaled_leaves(
...     staff.select_leaves(), tempo=tempo)
```

```
>>> for q_event in sequence:
...     print format(q_event, 'storage')
...
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(0, 1),
    pitches=(
        pitchtools.NamedPitch("c'"),
    ),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(2500, 3),
    pitches=(
        pitchtools.NamedPitch("d'"),
        pitchtools.NamedPitch("fs'"),
    ),
)
quantizationtools.SilentQEvent(
    offset=durationtools.Offset(4375, 3),
)
quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(5000, 3),
    pitches=(
        pitchtools.NamedPitch("gqs'"),
    ),
)
quantizationtools.TerminalQEvent(
    offset=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*)

True when q-event sequence contains *expr*. Otherwise false.

Returns boolean.

QEventSequence.**__eq__** (*expr*)

True when q-event sequence equals *expr*. Otherwise false.

Returns boolean.

`QEventSequence.__format__ (format_specification='')`

Formats q-event sequence.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

```
>>> print format(sequence)
quantizationtools.QEventSequence(
  (
    quantizationtools.PitchedQEvent(
      offset=durationtools.Offset(0, 1),
      pitches=(
        pitchtools.NamedPitch("c'"),
      ),
    ),
    quantizationtools.SilentQEvent(
      offset=durationtools.Offset(1000, 1),
    ),
    quantizationtools.PitchedQEvent(
      offset=durationtools.Offset(1500, 1),
      pitches=(
        pitchtools.NamedPitch("c'"),
      ),
    ),
    quantizationtools.SilentQEvent(
      offset=durationtools.Offset(2750, 1),
    ),
    quantizationtools.PitchedQEvent(
      offset=durationtools.Offset(3250, 1),
      pitches=(
        pitchtools.NamedPitch("c'"),
      ),
    ),
    quantizationtools.TerminalQEvent(
      offset=durationtools.Offset(4000, 1),
    ),
  )
)
```

Returns string.

`QEventSequence.__getitem__ (expr)`

Gets *expr* from q-event sequence.

Returns item.

`QEventSequence.__iter__ ()`

Iterates q-event sequence.

Yields items.

`QEventSequence.__len__ ()`

Length of q-event sequence.

Returns nonnegative integer.

`(AbjadObject).__ne__ (expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

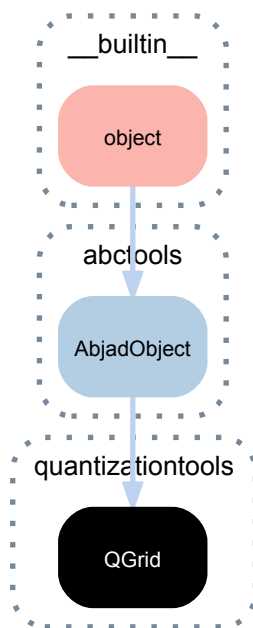
Returns boolean.

`(AbjadObject).__repr__ ()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.19 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()
```

```
>>> print format(q_grid, 'storage')
quantizationtools.QGrid(
  root_node=quantizationtools.QGridLeaf(
    preprolated_duration=durationtools.Duration(1, 1),
    is_divisible=True,
  ),
  next_downbeat=quantizationtools.QGridLeaf(
    preprolated_duration=durationtools.Duration(1, 1),
    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])
```

```
>>> for q_event_proxy in q_grid.root_node.q_event_proxies:
...     print format(q_event_proxy, 'storage')
...
quantizationtools.QEventProxy(
  q_event=quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(250, 1),
    pitches=(
      pitchtools.NamedPitch("c'"),
    ),
  ),
  offset=durationtools.Offset(1, 4),
)
```



```
>>> for q_event_proxy in q_grid.next_downbeat.q_event_proxies:
...     print format(q_event_proxy, 'storage')
...
quantizationtools.QEventProxy(
  q_event=quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(750, 1),
    pitches=(
      pitchtools.NamedPitch("cs'"),
    ),
  ),
  offset=durationtools.Offset(3, 4),
)
```

Used internally by the Quantizer.

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)`

Calls `q-grid`.

`QGrid.__copy__(*args)`

Copies `q-grid`.

Returns new `q-grid`.

`QGrid.__deepcopy__(memo)`

Deepcopies `q-grid`.

Returns new `q-grid`.

`QGrid.__eq__(expr)`

True if `expr` is a `q-grid` with root node and next downbeat equal to those of this `q-grid`. Otherwise false.

Returns boolean.

`QGrid.__format__(format_specification='')`

Formats `q-event`.

Set `format_specification` to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal `expr`. Otherwise false.

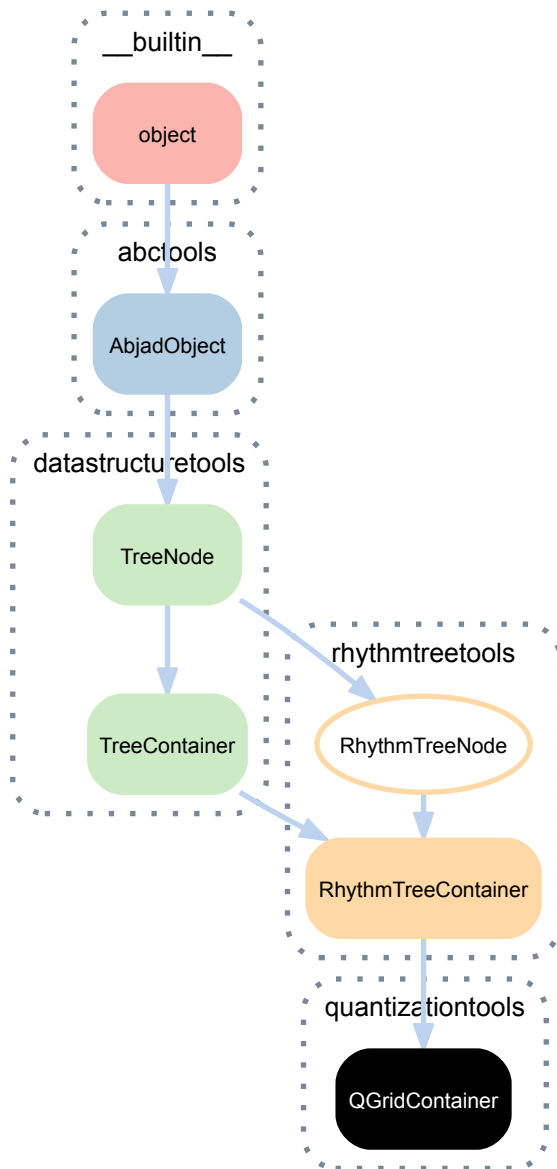
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.20 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`

Children of tree container.

```
>>> 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 tree nodes.

`(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 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**

Graph order of tree node.

Returns tuple.

(RhythmTreeNode) **.graphviz_format**

Graphviz format of rhythm tree node.

(RhythmTreeContainer) **.graphviz_graph**

The GraphvizGraph representation of the RhythmTreeContainer:

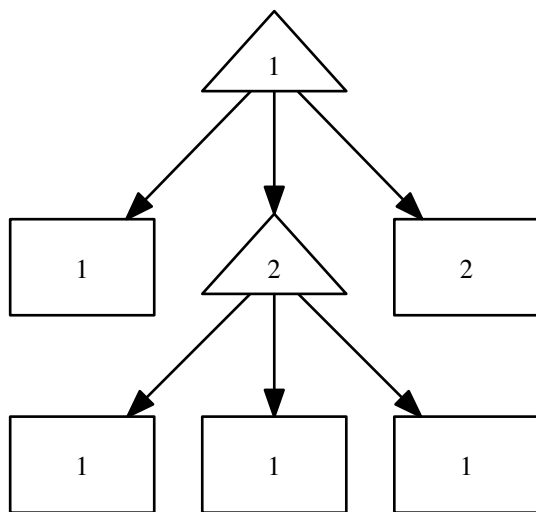
```

>>> 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;
}
```

```
>>> topleveltools.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 tree nodes.

(TreeContainer). **leaves**

Leaves of tree container.

```
>>> 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 tree nodes produced by iterating tree container 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

Parent of tree 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
```

Returns tree node.

(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 and the total preprolated_duration of that node and its siblings:

```
>>> a = rhythmreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmreetools.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**

Prolation of rhythm tree node.

Returns multiplier.

(RhythmTreeNode).**.prolations**

Prolations of rhythm tree node.

Returns tuple.

(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 tree nodes.

(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
```

Returns tree node.

(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

Named of tree node.

Returns string.

(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*)
 Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) . **extend** (*expr*)
 Extends *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) . **index** (*node*)
 Indexes *node* in tree 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 tree 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*)

Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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(Duration(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)
Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(*args)
Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
Deletes node *i* in tree container.

```
>>> 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.

(AbjadObject).**__format__**(*format_specification*='')
Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TreeContainer).__getitem__(i)`
 Gets node *i* in tree 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.

`(TreeContainer).__iter__()`
 Iterates tree container.

Yields children of tree container.

`(TreeContainer).__len__()`
 Returns nonnegative integer number of nodes in container.

`(TreeNode).__ne__(expr)`
 True when tree node does not equal *expr*. Otherwise false.
 Returns boolean.

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

`(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
```

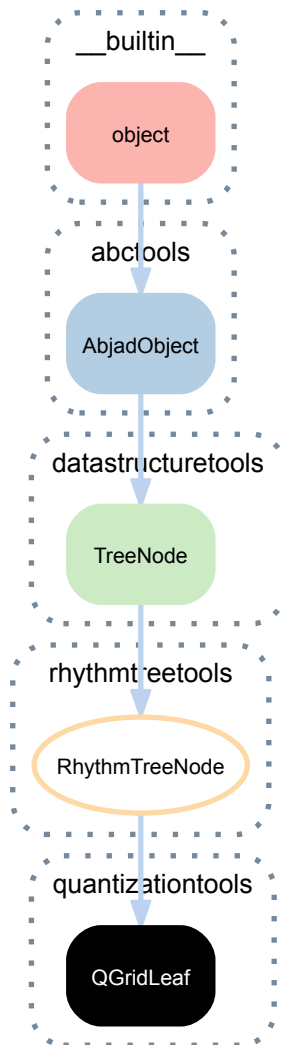
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

14.2.21 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.

Bases

- `rhythmtreertools.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 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**

Graph order of tree node.

Returns tuple.

(RhythmTreeNode) **.graphviz_format**

Graphviz format of rhythm tree node.

QGridLeaf **.graphviz_graph**

Graphviz graph of q-grid leaf.

Returns 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 tree nodes.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 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.

QGridLeaf **.preceding_q_event_proxies**

Preceding q-event proxies of q-grid leaf.

Returns list.

(RhythmTreeNode) **.pretty_rtm_format**

The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreertools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
  (1 (
    1
  ))
  (1 (
    1
  )))
)
```

Returns string.

(RhythmTreeNode) **.prolation**

Prolation of rhythm tree node.

Returns multiplier.

(RhythmTreeNode) **.prolations**

Prolations of rhythm tree node.

Returns tuple.

(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 tree nodes.

QGridLeaf **.q_event_proxies**

Q-event proxies of q-grid leaf.

(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
```

Returns tree node.

QGridLeaf **.rtm_format**

RTM format of q-grid leaf.

(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 = rhythmtreertools.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**

Succeeding q-event proxies of q-grid leaf.

Returns list.

Read/write properties

QGridLeaf **.is_divisible**

Flag for whether the node may be further divided under some search tree.

(TreeNode) **.name**

Named of tree node.

Returns string.

(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*)

Calls q-grid leaf.

Returns selection of notes.

(TreeNode) **.__copy__**(*args)

Copies tree node.

Returns new tree node.

QGridLeaf **.__deepcopy__**(memo)

Deepcopies q-grid leaf.

Returns new q-grid leaf.

QGridLeaf **.__eq__**(*expr*)

True when *expr* is a q-grid leaf with preprolated duration, q-event proxies and divisibility flag equal to those of this q-grid leaf. Otherwise false.

Returns boolean.

(AbjadObject) **.__format__**(*format_specification*='')

Formats object.

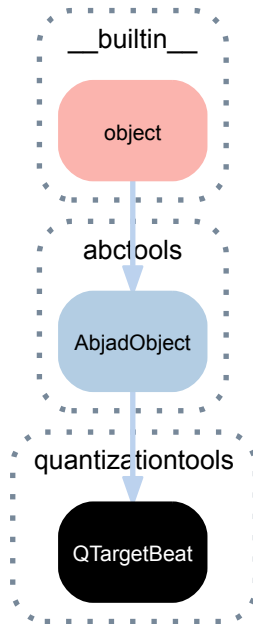
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TreeNode) .**__ne__**(*expr*)
 True when tree node does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject) .**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

14.2.22 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 = Tempo((1, 4), 56)
```

```
>>> q_target_beat = quantizationtools.QTargetBeat(
...     beatspan=beatspan,
...     offset_in_ms=offset_in_ms,
...     search_tree=search_tree,
...     tempo=tempo,
... )
```

```
>>> print format(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=indicatortools.Tempo(
        durationtools.Duration(1, 4),
        56
    ),
)
```

Not composer-safe.

Used internally by `Quantizer`.

Bases

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

Read-only properties

`QTargetBeat.beatspan`

Beatspan of q-target beat.

```
>>> 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`

Duration in milliseconds of the q-target beat.

```
>>> q_target_beat.duration_in_ms
Duration(3750, 7)
```

Returns duration.

`QTargetBeat.offset_in_ms`

Offset in milliseconds of q-target beat.

```
>>> q_target_beat.offset_in_ms
Offset(1500, 1)
```

Returns offset.

`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`.

Returns `QGrid` instance.

`QTargetBeat.q_grids`

A tuple of `QGrids` generated by a `QuantizationJob`.

Used internally by the `Quantizer`.

Returns tuple.

`QTargetBeat.search_tree`

Search tree of q-target beat.

```
>>> q_target_beat.search_tree
UnweightedSearchTree(definition={3: None})
```

Returns search tree.

`QTargetBeat`.**tempo**

Tempo of q-target beat.

```
>>> q_target_beat.tempo
Tempo(Duration(1, 4), 56)
```

Returns tempo.

Special methods

`QTargetBeat`.**__call__**(*job_id*)

Calls q-target beat.

Returns quantization job.

(`AbjadObject`).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`QTargetBeat`.**__format__**(*format_specification*='')

Formats q-event.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(`AbjadObject`).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

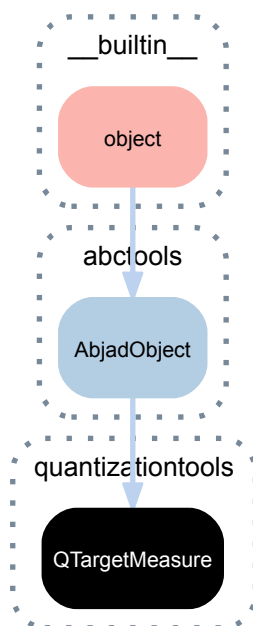
Returns boolean.

(`AbjadObject`).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.23 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 = Tempo((1, 4), 60)
>>> time_signature = TimeSignature((4, 4))
```

```
>>> q_target_measure = quantizationtools.QTargetMeasure(
...     offset_in_ms=1000,
...     search_tree=search_tree,
...     tempo=tempo,
...     time_signature=time_signature,
... )
```

```
>>> print format(q_target_measure, 'storage')
quantizationtools.QTargetMeasure(
  offset_in_ms=durationtools.Offset(1000, 1),
  search_tree=quantizationtools.UnweightedSearchTree(
    definition={ 2: None,
  },
),
  time_signature=indicatortools.TimeSignature(
    (4, 4)
  ),
  tempo=indicatortools.Tempo(
    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:
...     print format(q_target_beat, 'storage')
...
quantizationtools.QTargetBeat(
    beatspan=durationtools.Duration(1, 4),
    offset_in_ms=durationtools.Offset(1000, 1),
    search_tree=quantizationtools.UnweightedSearchTree(
        definition={ 2: None,
                    },
    ),
    tempo=indicatortools.Tempo(
        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=indicatortools.Tempo(
        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=indicatortools.Tempo(
        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=indicatortools.Tempo(
        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
Tempo(Duration(1, 4), 60)
```

Return `Tempo` instance.

`QTargetMeasure.time_signature`

The time signature of the `QTargetMeasure`:

```
>>> q_target_measure.time_signature
TimeSignature((4, 4))
```

Return `TimeSignature` 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)`

Is true when ID of `expr` equals ID of Abjad object. Otherwise false.

Returns boolean.

`QTargetMeasure.__format__(format_specification='')`

Formats q-event.

Set `format_specification` to `''` or `'storage'`. Interprets `''` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal `expr`. Otherwise false.

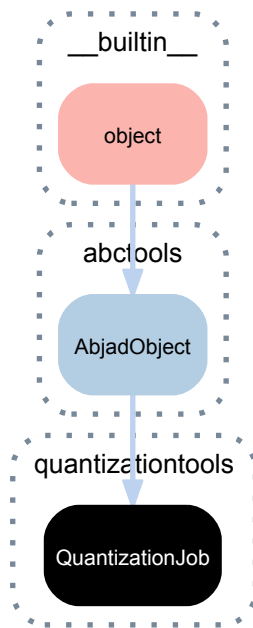
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.24 quantizationtools.QuantizationJob



class quantizationtools.**QuantizationJob** (*job_id=1*, *search_tree=None*,
q_event_proxies=None, *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:
...     print format(q_event_proxy, 'storage')
...
quantizationtools.QEventProxy(
  q_event=quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(250, 1),
    pitches=(
      pitchtools.NamedPitch("c'"),
      pitchtools.NamedPitch("cs'"),
    ),
  ),
  offset=durationtools.Offset(1, 4),
)
quantizationtools.QEventProxy(
  q_event=quantizationtools.SilentQEvent(
    offset=durationtools.Offset(500, 1),
  ),
  offset=durationtools.Offset(1, 2),
)
quantizationtools.QEventProxy(
  q_event=quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(750, 1),
    pitches=(
      pitchtools.NamedPitch("ef'"),
      pitchtools.NamedPitch("g'"),
    ),
  ),
  offset=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__()`

Calls quantization job.

Returns none.

`QuantizationJob.__eq__(expr)`

True when *expr* is a quantization job with job ID, search tree, q-event proxies and q-grids equal to those of this quantization job. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

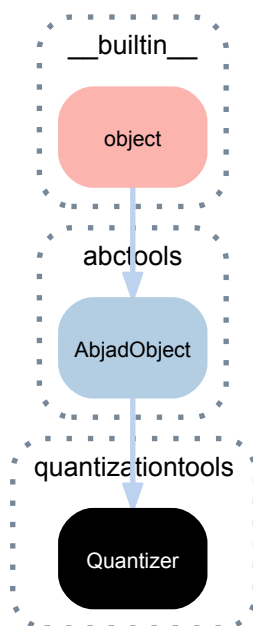
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.25 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])])
>>> print format(score)
\new Score <<
  \new Staff {
    \new Voice {
      {
        \tempo 4=60
        \time 4/4
        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])])
>>> print format(score)
\new Score <<
  \new Staff {
    \new Voice {
      {
        \tempo 4=78
        \time 2/4
        c'4 ~
        \times 4/5 {
          c'16.
          cs'8.. ~
        }
      }
      {
        \tempo 8=57
        \time 5/4
        \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 ~
        }
        r4
    }
}
>>

```

```
>>> 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)},
... })

```

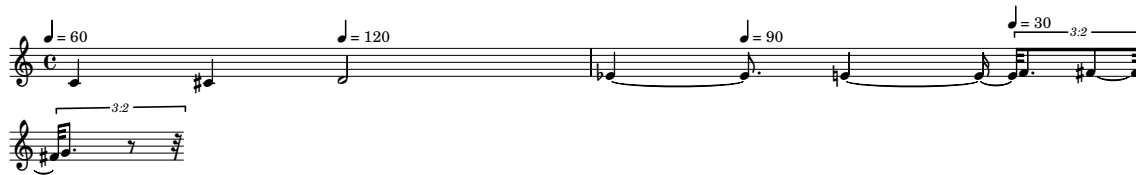
```

>>> result = quantizer(
...     q_event_sequence,
...     q_schema=beatwise_q_schema,
... )
>>> score = Score([Staff([result])])
>>> print format(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 logical ties are re-notated. Options currently include `MeasurewiseAttackPointOptimizer`, `NaiveAttackPointOptimizer` and `NullAttackPointOptimizer`.

Refer to the reference pages for `BeatwiseQSchema` and `MeasurewiseQSchema` for more information on controlling the Quantizer's output, and to the reference on `SearchTree` for information on controlling the rhythmic complexity of that same output.

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_tempos=True)`

Calls quantizer.

Returns Abjad components.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

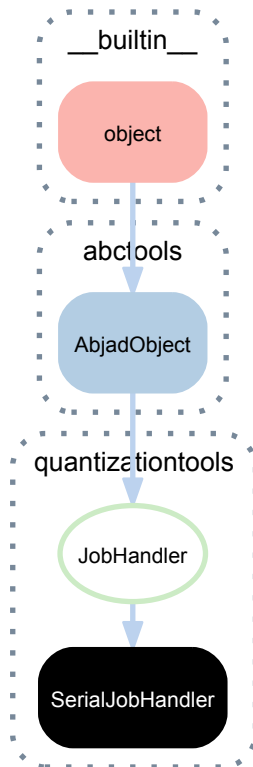
`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

14.2.26 quantizationtools.SerialJobHandler



class `quantizationtools.SerialJobHandler`
 Processes `QuantizationJob` instances sequentially.

Bases

- `quantizationtools.JobHandler`
- `abctools.AbjadObject`
- `__builtin__.object`

Special methods

`SerialJobHandler.__call__(jobs)`
 Calls serial job handler.
 Returns *jobs*.

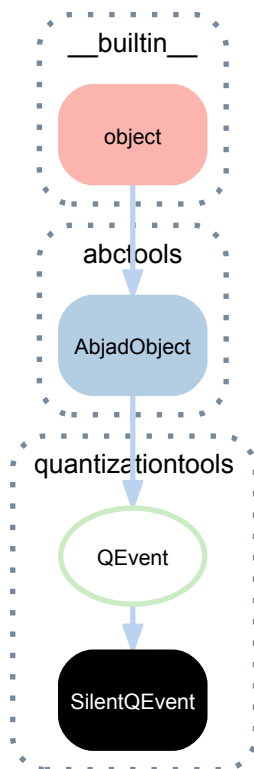
`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

14.2.27 quantizationtools.SilentQEvent



class quantizationtools.**SilentQEvent** (*offset=0, 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
SilentQEvent(offset=Offset(1000, 1))
    
```

Bases

- quantizationtools.QEvent
- abctools.AbjadObject
- __builtin__.object

Read-only properties

SilentQEvent.**attachments**
 Attachments of silen q-event.

(QEvent).**index**
 The optional index, for sorting QEvents with identical offsets.

`(QEvent).offset`

The offset in milliseconds of the event.

Special methods

`SilentQEvent.__eq__(expr)`

True when *expr* is a silent q-event with offset, attachments and index equal to those of this silent q-event. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(QEvent).__lt__(expr)`

True when *expr* is a q-event with offset greater than that of this q-event. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

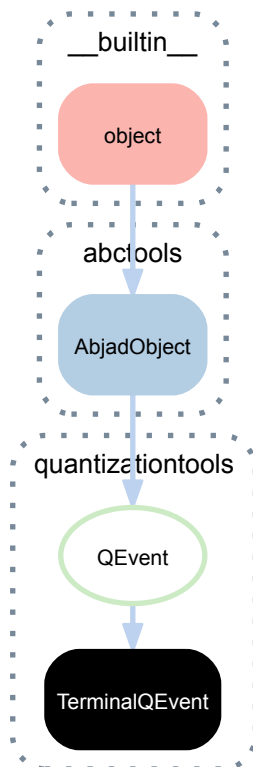
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.28 quantizationtools.TerminalQEvent



class `quantizationtools.TerminalQEvent` (*offset=0*)

The terminal event in a series of `QEvents`:

```
>>> q_event = quantizationtools.TerminalQEvent(1000)
>>> print format(q_event)
quantizationtools.TerminalQEvent(
    offset=durationtools.Offset(1000, 1),
)
```

Carries no significance outside the context of a `QEventSequence`.

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.

Special methods

`TerminalQEvent.__eq__(expr)`

True when *expr* is a terminal q-event with offset equal to that of this terminal q-event. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `''` or `'storage'`. Interprets `''` equal to `'storage'`.

Returns string.

`(QEvent).__lt__(expr)`

True when *expr* is a q-event with offset greater than that of this q-event. Otherwise false.

Returns boolean.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

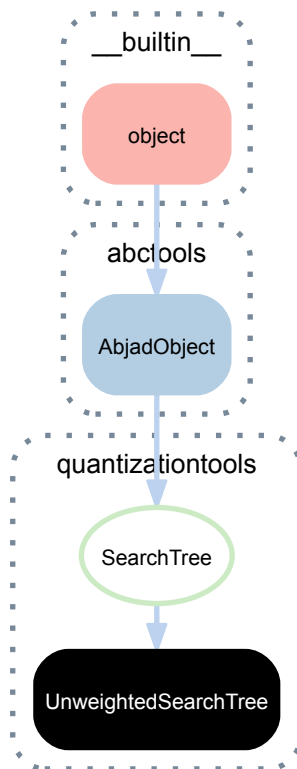
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.29 quantizationtools.UnweightedSearchTree



class `quantizationtools.UnweightedSearchTree` (*definition=None*)
 Concrete SearchTree subclass, based on Paul Nauert's search tree model:

```

>>> search_tree = quantizationtools.UnweightedSearchTree()
>>> print format(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,
  },
)
  
```

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*)

Calls search tree.

(SearchTree).**__eq__**(*expr*)

True when *expr* is a search tree with definition equal to that of this search tree. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

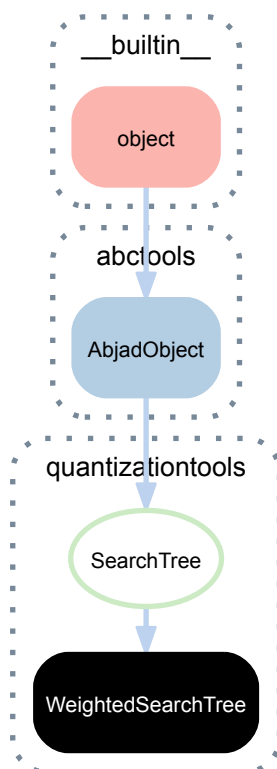
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.30 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()
```

```
>>> print format(search_tree)
quantizationtools.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)
```

Bases

- `quantizationtools.SearchTree`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`WeightedSearchTree.all_compositions`

All compositions of weighted search tree.

`WeightedSearchTree.default_definition`

Default definition of weighted search tree.

Returns dictionary.

`(SearchTree).definition`

The search tree definition.

Returns dictionary.

Special methods

(SearchTree) .**__call__**(*q_grid*)

Calls search tree.

(SearchTree) .**__eq__**(*expr*)

True when *expr* is a search tree with definition equal to that of this search tree. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.3 Functions

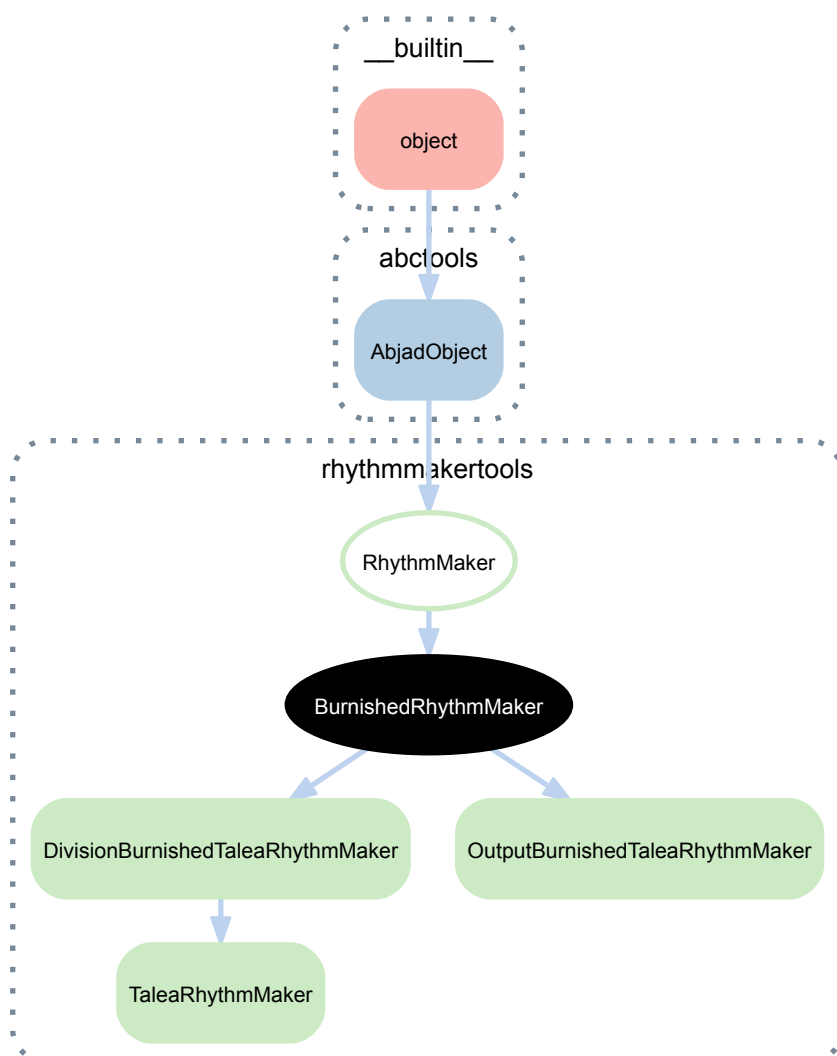
14.3.1 quantizationtools.make_test_time_segments

quantizationtools.**make_test_time_segments**()

Make test time segments.

15.1 Abstract classes

15.1.1 `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.AbjadObject`
- `__builtin__.object`

Methods

`BurnishedRhythmMaker.reverse()`

Reverses 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)`

Calls 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.

`BurnishedRhythmMaker.__format__` (*format_specification*='')

Formats burnished rhythm-maker.

Set *format_specification* to '' or 'storage'.

Returns string.

`(RhythmMaker).__makenew__` (*args, **kwargs)

Creates new rhythm-maker with *kwargs*.

```
>>> maker = rhythmtools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = new(maker, decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(
...     scoretools.make_spacer_skip_measures(
...         divisions))
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

`(AbjadObject).__ne__` (*expr*)

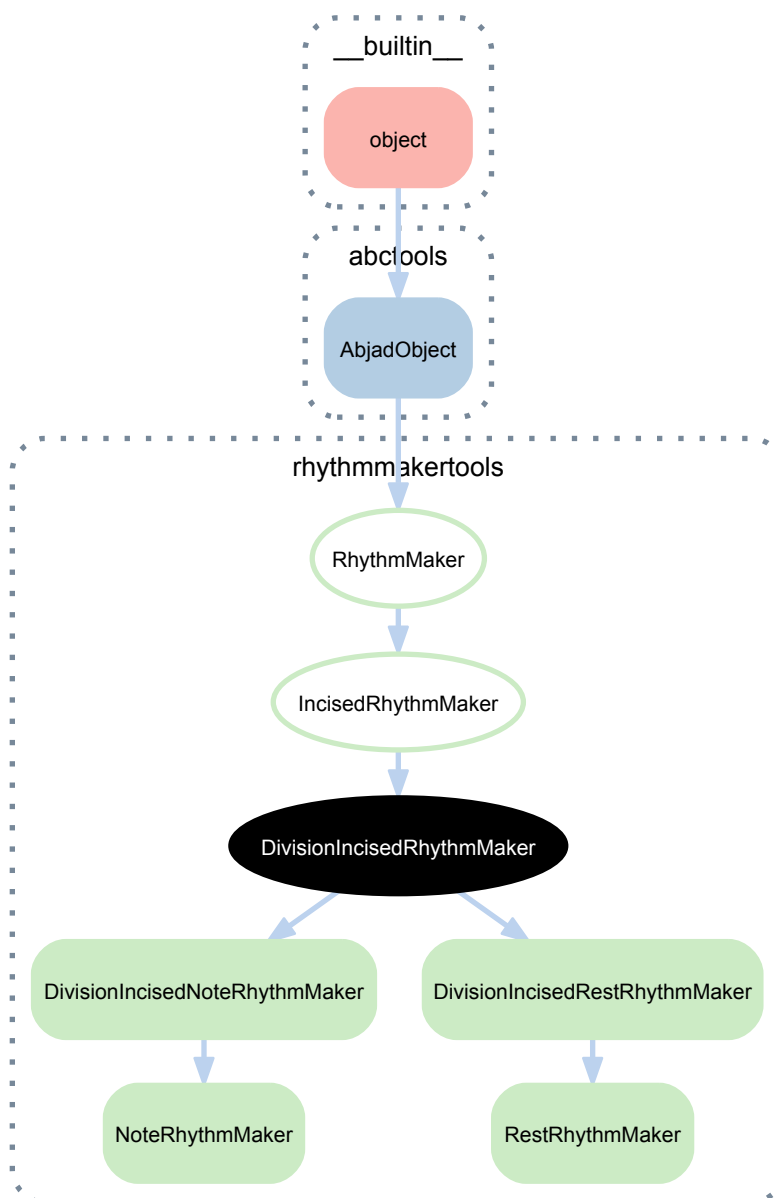
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__` ()

Gets interpreter representation of Abjad object.

Returns string.

15.1.2 `rhythmmakertools.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.AbjadObject`
- `__builtin__.object`

Methods

(IncisedRhythmMaker).**reverse**()
 Reverses incised rhythm-maker.
 Returns newly constructed rhythm-maker.

Special methods

(IncisedRhythmMaker).**__call__**(*divisions*, *seeds*=None)
 Calls 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.

(RhythmMaker).**__format__**(*format_specification*='')
 Formats rhythm-maker.
 Set *format_specification* to '' or 'storage'.
 Defaults *format_specification*=None to *format_specification*='storage'.
 Returns string.

(RhythmMaker).**__makenew__**(*args, **kwargs)
 Creates new rhythm-maker with *kwargs*.

```
>>> maker = rhythmmakertools.NoteRhythmMaker()

>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = new(maker, decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)

>>> staff = Staff(
...     scoretools.make_spacer_skip_measures(
...         divisions))
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

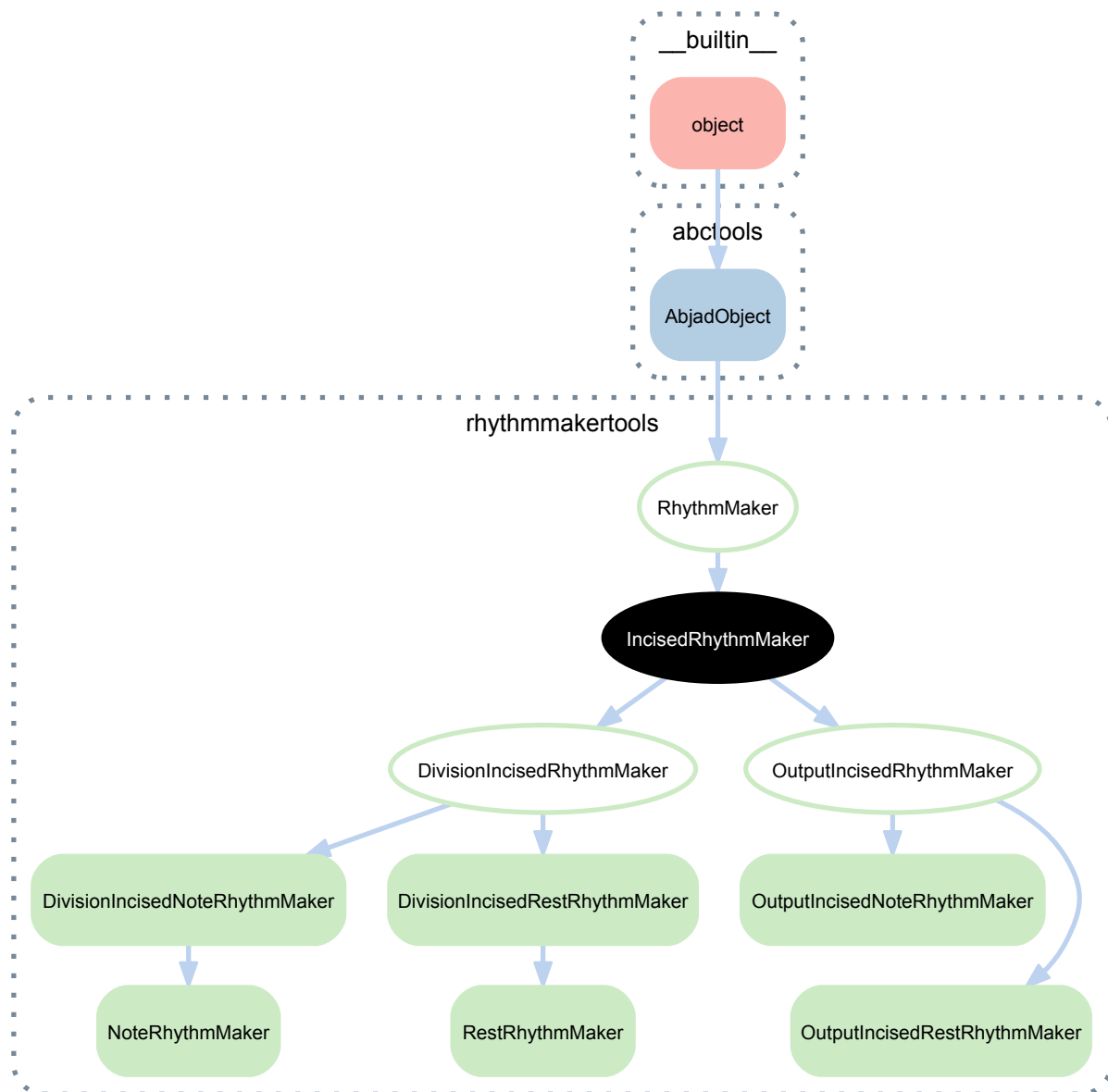
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.1.3 `rhythmmakertools.IncisedRhythmMaker`



```
class rhythmmakertools.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,
                                             prefix_talea_helper=None, prefix_lengths_helper=None,
                                             suffix_talea_helper=None, suffix_lengths_helper=None,
                                             prolation_addenda_helper=None, secondary_divisions_helper=None,
                                             decrease_durations_monotonically=True,
                                             tie_rests=False, forbidden_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.AbjadObject`
- `__builtin__.object`

Methods

`IncisedRhythmMaker.reverse()`
 Reverses incised rhythm-maker.
 Returns newly constructed rhythm-maker.

Special methods

`IncisedRhythmMaker.__call__(divisions, seeds=None)`
 Calls 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.

`(RhythmMaker).__format__(format_specification='')`
 Formats rhythm-maker.
 Set *format_specification* to `'` or `'storage'`.
 Defaults *format_specification=None* to *format_specification='storage'*.
 Returns string.

`(RhythmMaker).__makenew__(*args, **kwargs)`
 Creates new rhythm-maker with *kwargs*.

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = new(maker, decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

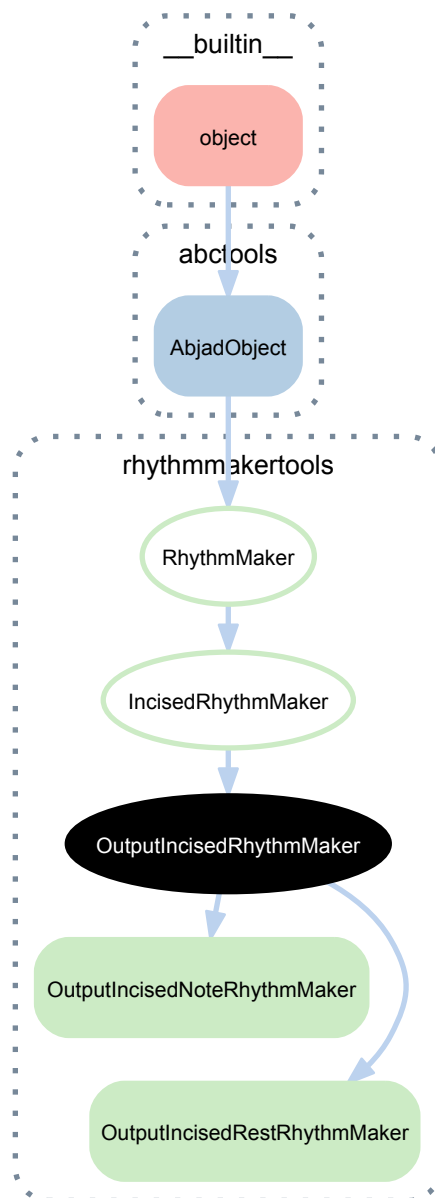
```
>>> staff = Staff(
...     scoretools.make_spacer_skip_measures(
...         divisions))
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

15.1.4 `rhythmmakertools.OutputIncisedRhythmMaker`



```

class rhythmmakertools.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

- `rhythmmakertools.IncisedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadObject`
- `__builtin__.object`

Methods

`(IncisedRhythmMaker).reverse()`
 Reverses incised rhythm-maker.
 Returns newly constructed rhythm-maker.

Special methods

`(IncisedRhythmMaker).__call__(divisions, seeds=None)`
 Calls 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.

`(RhythmMaker).__format__(format_specification='')`
 Formats rhythm-maker.
 Set *format_specification* to `'` or `'storage'`.
 Defaults *format_specification=None* to *format_specification='storage'*.
 Returns string.

`(RhythmMaker).__makenew__(*args, **kwargs)`
 Creates new rhythm-maker with *kwargs*.

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = new(maker, decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

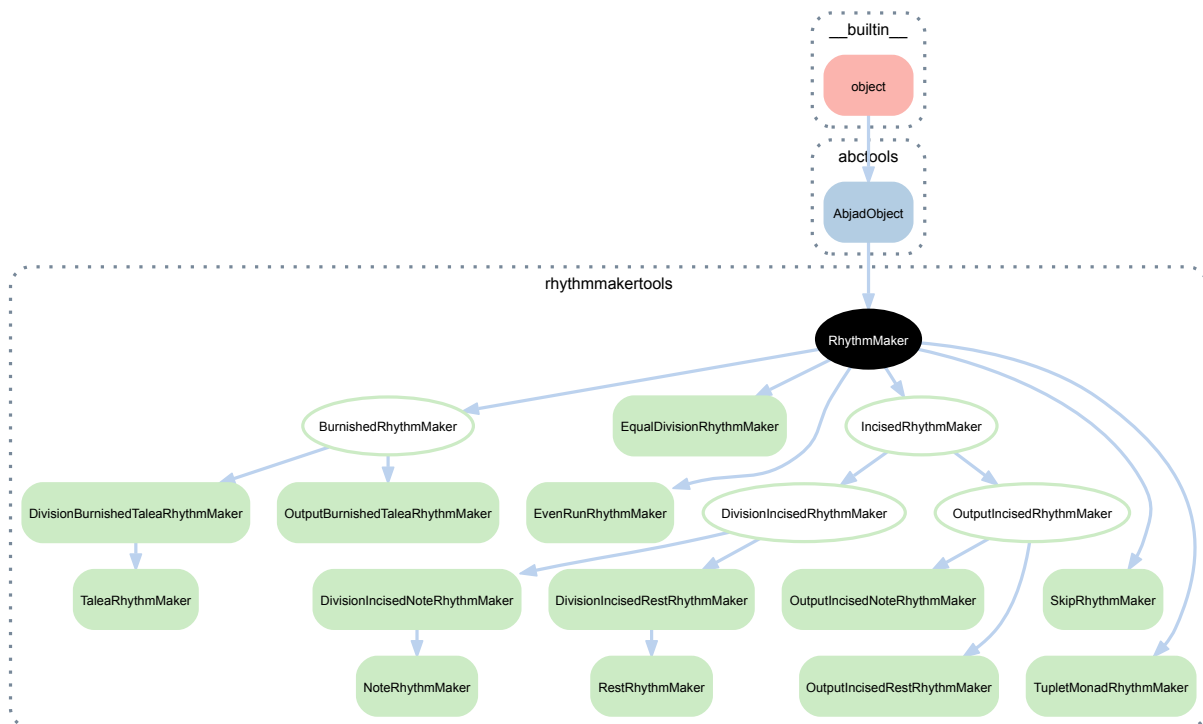
```
>>> staff = Staff(
...     scoretools.make_spacer_skip_measures(
...         divisions))
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

15.1.5 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.AbjadObject`
- `__builtin__.object`

Methods

`RhythmMaker.reverse()`
 Reverses 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)`
 Casts *divisions* into duration pairs. Reduces numerator and denominator relative to each other.
 Changes 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.

`RhythmMaker.__format__(format_specification='')`

Formats rhythm-maker.

Set *format_specification* to '' or 'storage'.

Defaults *format_specification*=None to *format_specification*='storage'.

Returns string.

`RhythmMaker.__makenew__(*args, **kwargs)`

Creates new rhythm-maker with *kwargs*.

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> new_maker = new(maker, decrease_durations_monotonically=False)
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(
...     scoretools.make_spacer_skip_measures(
...         divisions))
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

Returns new rhythm-maker.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

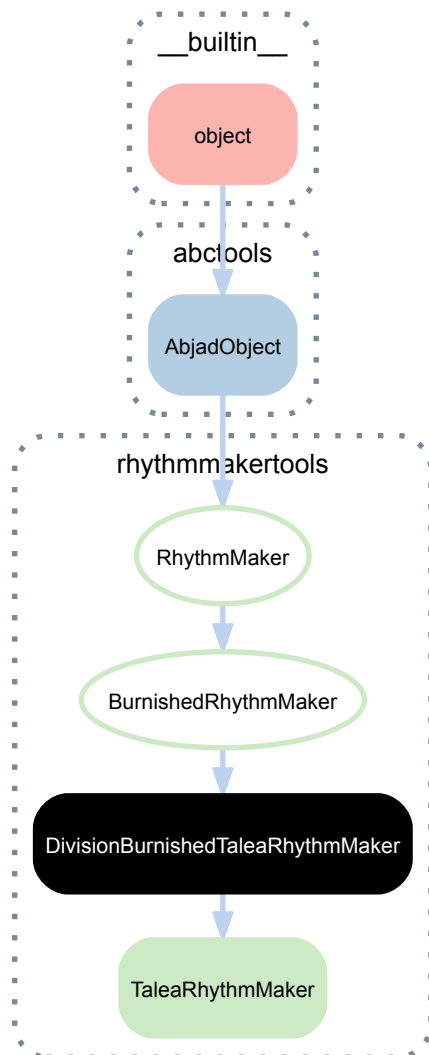
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

15.2 Concrete classes

15.2.1 `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)
```

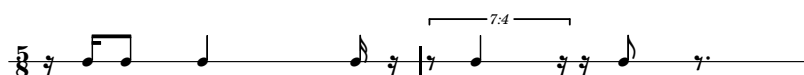
Division-burnished talea rhythm-maker:

```
>>> 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 = \
...     scoretools.make SpacerSkipMeasures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Methods

`DivisionBurnishedTaleaRhythmMaker.reverse()`

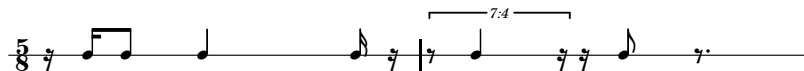
Reverses division-burnished talea rhythm-maker.

Nonreversed output:

```
>>> print format(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],
    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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)

>>> show(staff)
```



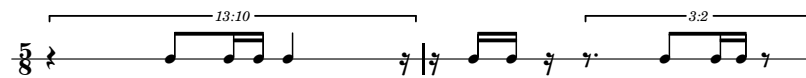
Reversed output:

```
>>> reversed_maker = maker.reverse()

>>> print format(reversed_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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



Returns new division-burnished talea rhythm-maker.

Special methods

`(BurnishedRhythmMaker).__call__(divisions, seeds=None)`

Calls 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.

`DivisionBurnishedTaleaRhythmMaker.__format__(format_specification='')`

Formats division-burnished talea rhythm-maker.

Set *format_specification* to ‘’ or ‘storage’.

```
>>> print format(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],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False,
)
```

Returns string.

`DivisionBurnishedTaleaRhythmMaker.__makenew__(*args, **kwargs)`

Creates new rhythm-maker with *kwargs*.

```
>>> print format(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],
    beam_each_cell=False,
```

```

beam_cells_together=False,
decrease_durations_monotonically=True,
tie_split_notes=False,
tie_rests=False,
)

```

```
>>> new_maker = new(maker, talea=[1, 1, 2])
```

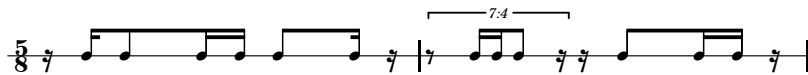
```
>>> print format(new_maker)
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
  talea=[1, 1, 2],
  talea_denominator=16,
  prolation_addenda=[0, 3],
  lefts=[-1],
  middles=[0],
  rights=[-1],
  left_lengths=[1],
  right_lengths=[1],
  secondary_divisions=[14],
  beam_each_cell=False,
  beam_cells_together=False,
  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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)

```

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



Returns new division-burnished talea rhythm-maker.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

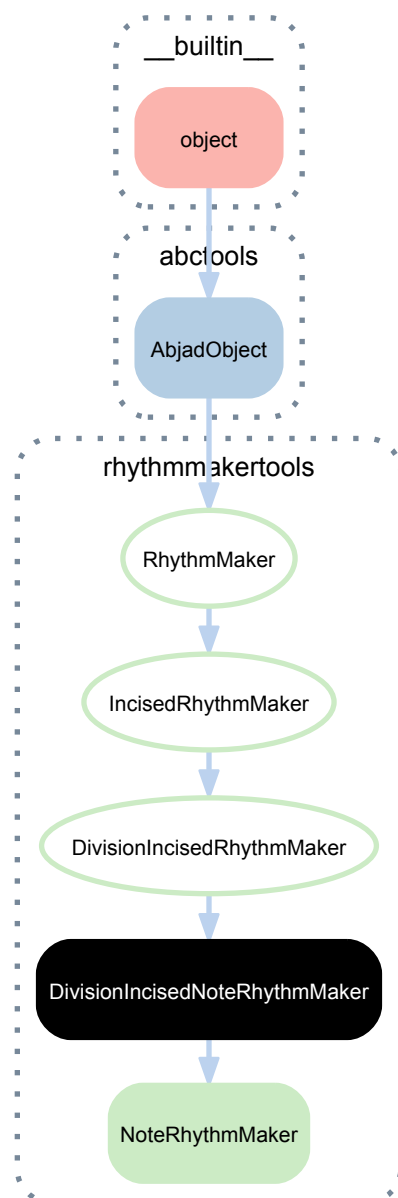
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.2 `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:

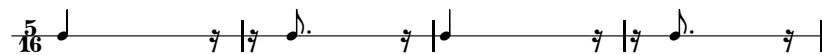
Example 1. Basic usage:

```
>>> 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 = \
...     scoretools.make_spacer_skip_measures (
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr (
...     staff, leaves)
```

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



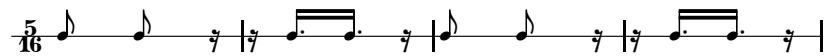
Example 2. Set *body_ratio* to divide middle part proportionally:

```
>>> 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 = \
...     scoretools.make_spacer_skip_measures (
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
```

```
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Methods

`DivisionIncisedNoteRhythmMaker.reverse()`

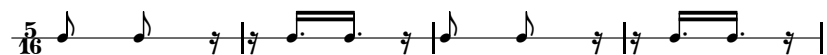
Reverses division-incised note rhythm-maker.

Nonreversed output:

```
>>> print format(maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

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



Reversed output:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)
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,
)

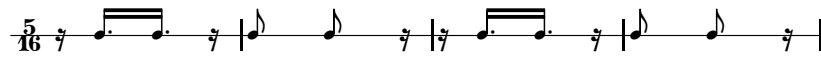
```

```

>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)

```

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



Returns division-incised note rhythm-maker.

Special methods

(IncisedRhythmMaker).**__call__**(*divisions*, *seeds=None*)

Calls 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.

DivisionIncisedNoteRhythmMaker.**__format__**(*format_specification=''*)

Formats division-incised note rhythm-maker.

Set *format_specification* to `'` or `'storage'`.

```

>>> print format(maker)
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.

DivisionIncisedNoteRhythmMaker.**__makenew__**(**args*, ***kwargs*)

Creates new division-incised note rhythm-maker with *kwargs*.

```

>>> print format(maker)
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 = new(maker, prefix_lengths=[1])
```

```

>>> print format(new_maker)
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,
)

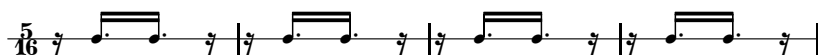
```

```

>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)

```

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



Returns new division-incised note rhythm-maker.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

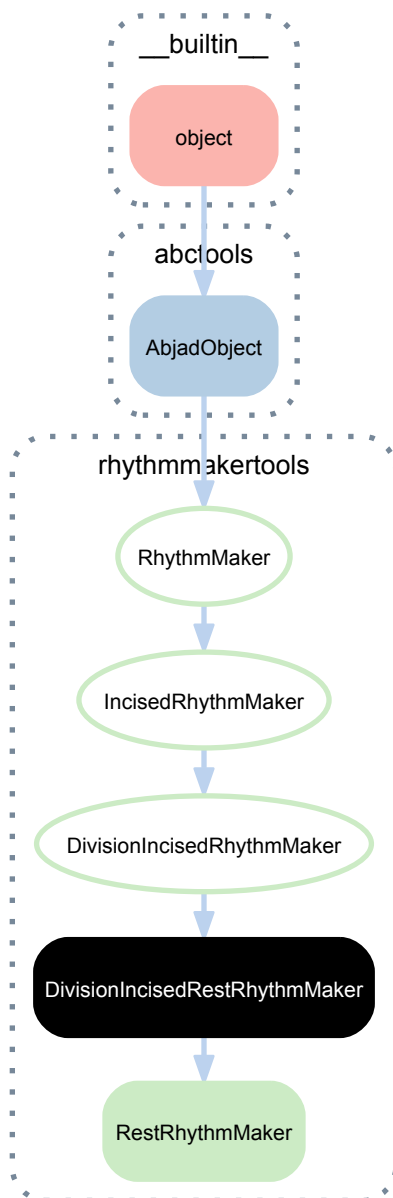
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.3 rhythm makertools.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 = \
...     scoretools.make_spacer_skip_measures (
...         divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Methods

`DivisionIncisedRestRhythmMaker.reverse()`

Reverses division-incised rest rhythm-maker.

```
>>> print format(maker)
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 format(reversed_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

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



Returns new division-incised rest rhythm-maker.

Special methods

`(IncisedRhythmMaker).__call__(divisions, seeds=None)`

Calls 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.

`DivisionIncisedRestRhythmMaker.__format__(format_specification='')`

Formats division-incised rest rhythm-maker.

Set *format_specification* to `'` or `'storage'`.

```
>>> print format(maker)
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.

`DivisionIncisedRestRhythmMaker.__makenew__(*args, **kwargs)`

Creates new division-incised rest rhythm-maker with *kwargs*.

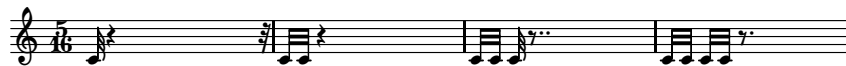
```
>>> print format(maker)
rhythmmakertools.DivisionIncisedRestRhythmMaker(
  prefix_talea=[1],
  prefix_lengths=[1, 2, 3, 4],
  suffix_talea=[1],
  suffix_lengths=[1],
  talea_denominator=32,
  prolation_addenda=[],
  secondary_divisions=[],
  decrease_durations_monotonically=True,
  tie_rests=False,
  beam_each_cell=False,
  beam_cells_together=False,
)
```

```
>>> new_maker = new(maker, suffix_lengths=[0])
```

```
>>> print format(new_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

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

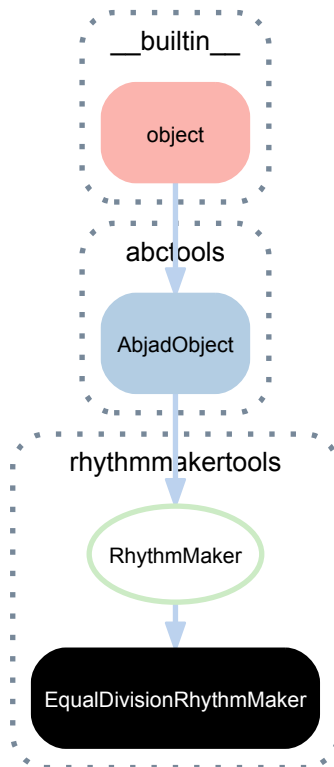


Returns new division-incised rest rhythm-maker.

(AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

15.2.4 `rhythmmakertools.EqualDivisionRhythmMaker`



```
class rhythmmakertools.EqualDivisionRhythmMaker(leaf_count=1, 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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.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.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.

Methods

`EqualDivisionRhythmMaker.reverse()`

Reverses equal-division rhythm-maker.

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.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)`

Calls 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.

`EqualDivisionRhythmMaker.__format__ (format_specification='')`

Formats equal division rhythm-maker.

Set *format_specification* to `'` or `'storage'`.

```
>>> print format(maker)
rhythmmakertools.EqualDivisionRhythmMaker(
  leaf_count=4,
  is_diminution=True,
  beam_each_cell=True,
  beam_cells_together=False,
)
```

Returns string.

`EqualDivisionRhythmMaker.__makenew__ (*args, **kwargs)`

Creates new equal-division rhythm-maker with *kwargs*.

```
>>> new_maker = new(maker, is_diminution=False)
```

```
>>> print format(new_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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



Returns new equal-division rhythm-maker.

`(AbjadObject).__ne__ (expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

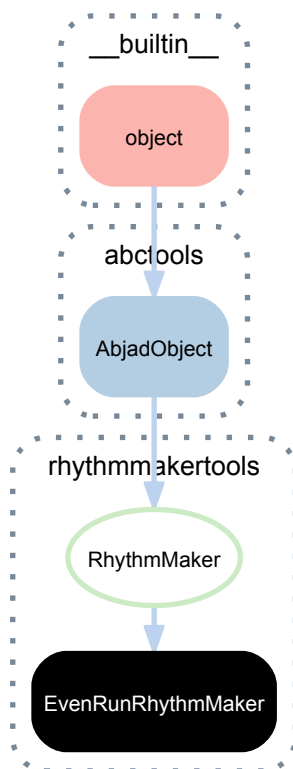
Returns boolean.

`(AbjadObject).__repr__ ()`

Gets interpreter representation of Abjad object.

Returns string.

15.2.5 `rhythmmakertools.EvenRunRhythmMaker`



class `rhythmmakertools.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 = rhythmmakertools.EvenRunRhythmMaker()
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.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:

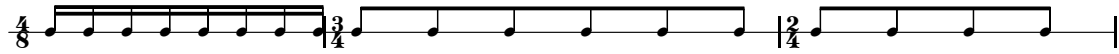
```
>>> maker = rhythmmakertools.EvenRunRhythmMaker(1)
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
```



```
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Read-only properties

`EvenRunRhythmMaker.denominator_multiplier_exponent`
Denominator multiplier exponent provided at initialization.

```
>>> maker.denominator_multiplier_exponent
1
```

Returns nonnegative integer.

Methods

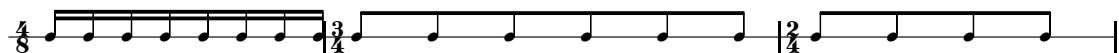
`EvenRunRhythmMaker.reverse()`
Reverses even-run rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.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)`

Calls 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.

`EvenRunRhythmMaker.__format__ (format_specification='')`

Formats even run rhythm-maker.

Set *format_specification* to `'` or `'storage'`.

```
>>> print format(maker)
rhythmmakertools.EvenRunRhythmMaker(
  denominator_multiplier_exponent=1,
  beam_each_cell=True,
  beam_cells_together=False,
)
```

Returns string.

`EvenRunRhythmMaker.__makenew__ (*args, **kwargs)`

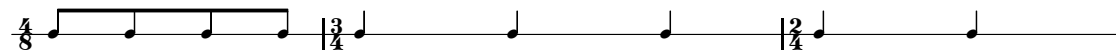
Creates new even-run rhythm-maker with *kwargs*.

```
>>> new_maker = new(maker, denominator_multiplier_exponent=0)
```

```
>>> print format(new_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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



Returns new even-run rhythm-maker.

`(AbjadObject).__ne__ (expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

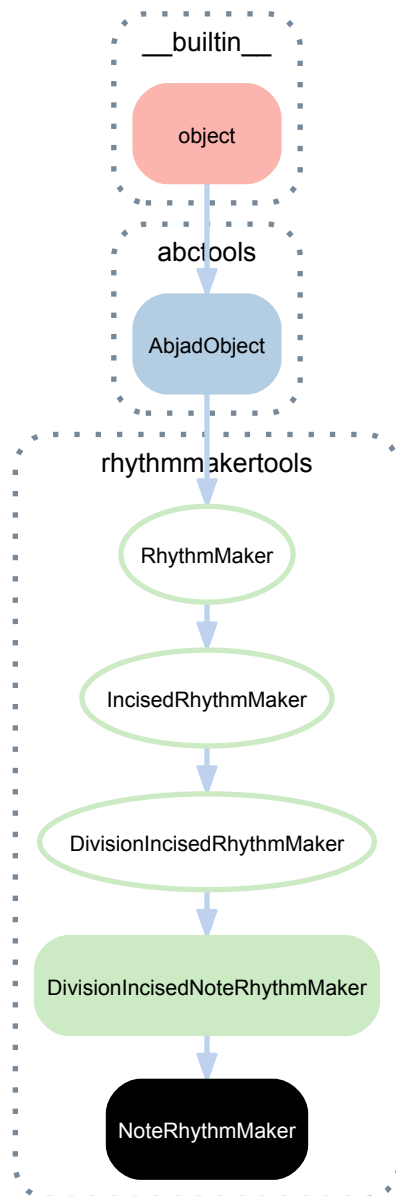
Returns boolean.

`(AbjadObject).__repr__ ()`

Gets interpreter representation of Abjad object.

Returns string.

15.2.6 rhythm makertools.NoteRhythmMaker



class `rhythm makertools.NoteRhythmMaker` (*decrease_durations_monotonically=True, forbidden_written_duration=None, tie_rests=False*)

Note rhythm-maker:

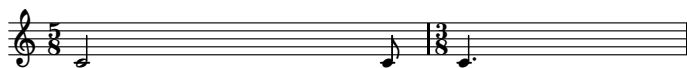
Example 1:

```

>>> maker = rhythm makertools.NoteRhythmMaker()

>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)

>>> show(staff)
  
```

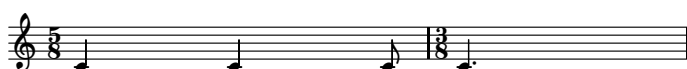


Example 2. Forbid half notes:

```
>>> maker = rhythmmakertools.NoteRhythmMaker(  
...     forbidden_written_duration=Duration(1, 2))
```

```
>>> divisions = [(5, 8), (3, 8)]  
>>> leaf_lists = maker(divisions)  
>>> leaves = sequencetools.flatten_sequence(leaf_lists)  
>>> measures = \  
...     scoretools.make_spacer_skip_measures(  
...         divisions)  
>>> staff = Staff(measures)  
>>> measures = scoretools.replace_contents_of_measures_in_expr(  
...     staff, leaves)
```

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



Usage follows the two-step instantiate-then-call pattern shown here.

Bases

- `rhythmmakertools.DivisionIncisedNoteRhythmMaker`
- `rhythmmakertools.DivisionIncisedRhythmMaker`
- `rhythmmakertools.IncisedRhythmMaker`
- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadObject`
- `__builtin__.object`

Methods

`NoteRhythmMaker.reverse()`

Reverses note rhythm-maker.

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)  
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 = \  
...     scoretools.make_spacer_skip_measures(  
...         divisions)  
>>> staff = Staff(measures)  
>>> measures = scoretools.replace_contents_of_measures_in_expr(  
...     staff, leaves)
```

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



Returns new note rhythm-maker.

Special methods

(IncisedRhythmMaker).**__call__**(*divisions*, *seeds=None*)

Calls 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.

NoteRhythmMaker.**__format__**(*format_specification=''*)

Formats note rhythm-maker.

Set *format_specification* to `'` or `'storage'`.

```
>>> print format(maker)
rhythm makertools.NoteRhythmMaker(
  decrease_durations_monotonically=True,
  forbidden_written_duration=durationtools.Duration(1, 2),
  tie_rests=False,
)
```

Returns string.

NoteRhythmMaker.**__makenew__**(**args*, ***kwargs*)

Creates new note rhythm-maker.

```
>>> new_maker = new(maker, decrease_durations_monotonically=False)
```

```
>>> print format(new_maker)
rhythm makertools.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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

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



Returns new note rhythm-maker.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

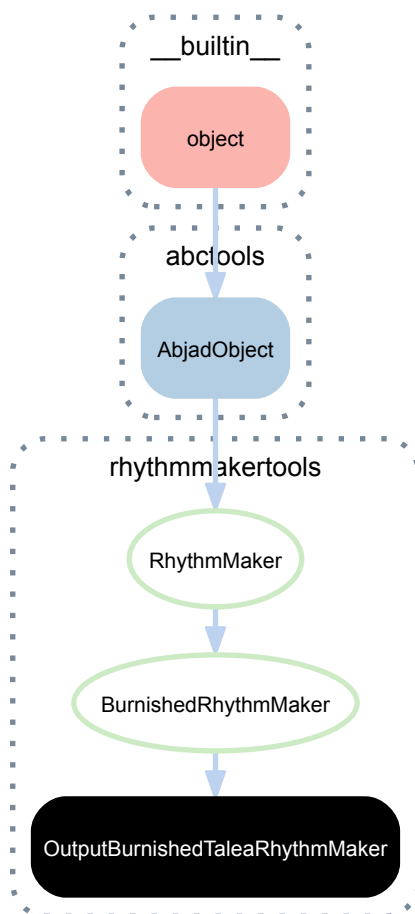
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.7 `rhythmmakertools.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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Methods

`OutputBurnishedTaleaRhythmMaker.reverse()`

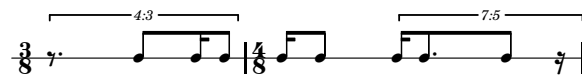
Reverses output-burnished talea rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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



Returns new output-burnished talea rhythm-maker.

Special methods

`(BurnishedRhythmMaker).__call__(divisions, seeds=None)`

Calls 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.

`OutputBurnishedTaleaRhythmMaker.__format__(format_specification='')`

Formats output-burnished talea rhythm-maker.

Set *format_specification* to '' or 'storage'.

```
>>> print format(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,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False,
)
```

Returns string.

`OutputBurnishedTaleaRhythmMaker.__makenew__(*args, **kwargs)`

Creates new output-burnished talea rhythm-maker with *kwargs*.

```
>>> new_maker = new(maker, secondary_divisions=[10])
```

```
>>> print format(new_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=[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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)

```

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



Returns new output-burnished talea rhythm-maker.

(AbjadObject).**__ne__**(*expr*)

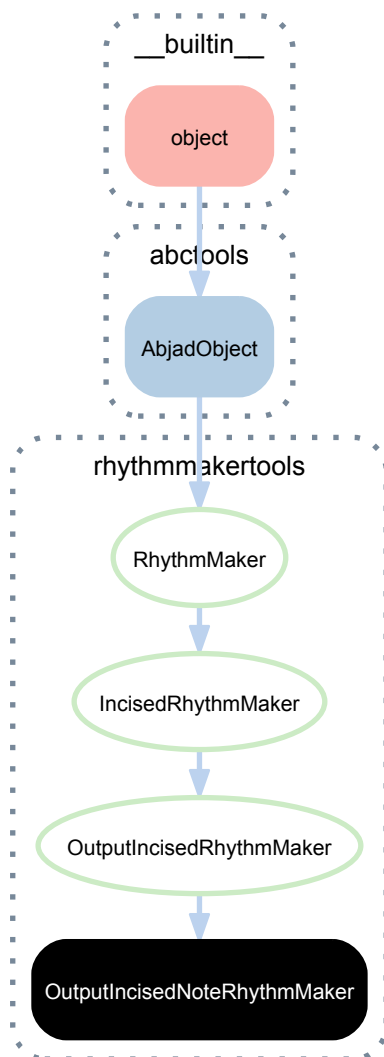
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.8 `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, 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)

```

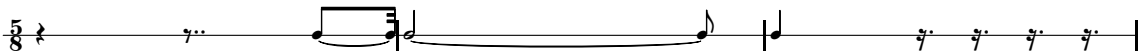
Output-incised note rhythm-maker:

```
>>> 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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Methods

`OutputIncisedNoteRhythmMaker.reverse()`

Reverses output-incised note rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)
rhythmmakertools.OutputIncisedNoteRhythmMaker(
    prefix_talea=[-7, -8],
    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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

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



Returns new output-incised note rhythm-maker.

Special methods

(IncisedRhythmMaker).**__call__**(*divisions*, *seeds=None*)

Calls 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.

OutputIncisedNoteRhythmMaker.**__format__**(*format_specification=''*)

Formats output-incised note rhythm-maker.

Set *format_specification* to `'` or `'storage'`.

```
>>> print format(maker)
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.

OutputIncisedNoteRhythmMaker.**__makenew__**(**args*, ***kwargs*)

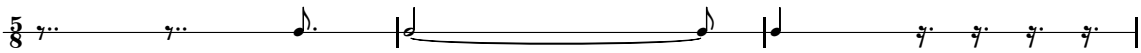
Creates new output-incised note rhythm-maker with *kwargs*.

```
>>> new_maker = new(maker, prefix_talea=[-7])
```

```
>>> print format(new_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

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



Returns new output-incised note rhythm-maker.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

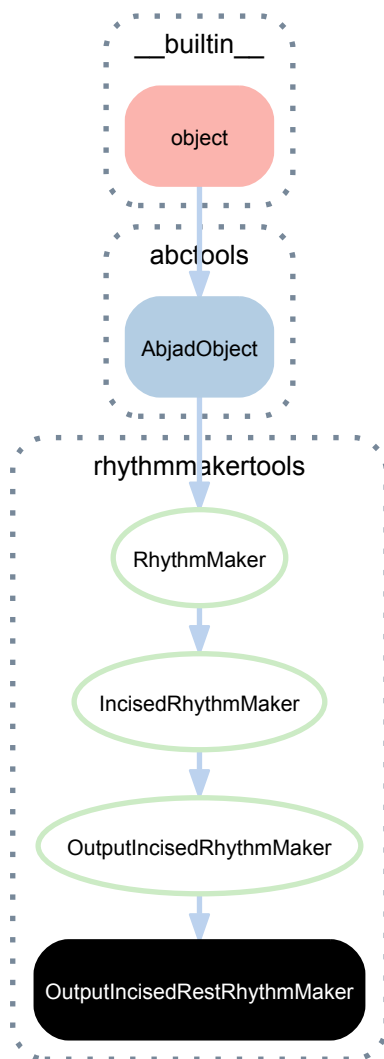
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.9 rhythm makertools.OutputIncisedRestRhythmMaker



```
class rhythmmakertools.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)
```

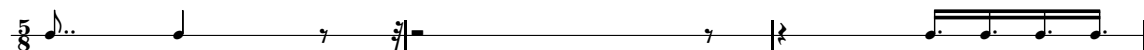
Output-incised rest rhythm-maker:

```
>>> 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 = \
...     scoretools.make_spacer_skip_measures(
...     divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Methods

`OutputIncisedRestRhythmMaker.reverse()`
Reverses output-incised rest rhythm-maker:


```
>>> new_maker = new(maker, prefix_talea=[7])
```

```
>>> print format(new_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

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



Returns new output-incised rest rhythm-maker.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

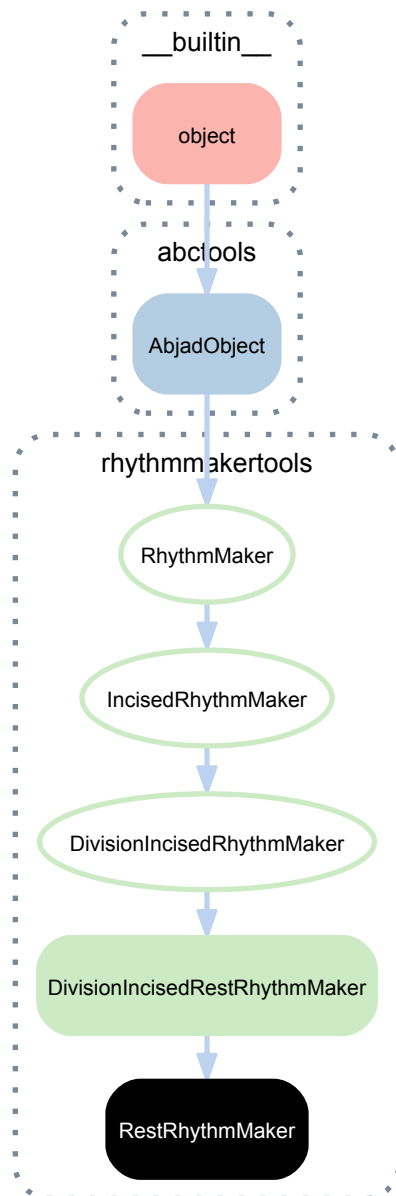
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.10 rhythmtools.RestRhythmMaker



class `rhythmtools.RestRhythmMaker` (*forbidden_written_duration=None*)
 Rest rhythm-maker.

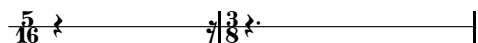
Example 1:

```
>>> maker = rhythmtools.RestRhythmMaker()
```

Initialize and then call on arbitrary divisions:

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

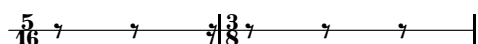
```
>>> show(staff)
```



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 = \  
...     scoretools.make_spacer_skip_measures(  
...         divisions)  
>>> staff = scoretools.RhythmicStaff(measures)  
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Methods

`RestRhythmMaker.reverse()`

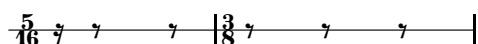
Reverses rest rhythm-maker.

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)  
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 = \  
...     scoretools.make_spacer_skip_measures(  
...         divisions)  
>>> staff = scoretools.RhythmicStaff(measures)  
>>> measures = scoretools.replace_contents_of_measures_in_expr(  
...     staff, leaves)
```

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



Returns new rest rhythm-maker.

Special methods

(IncisedRhythmMaker).**__call__**(*divisions*, *seeds=None*)

Calls 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.

RestRhythmMaker.**__format__**(*format_specification=''*)

Formats rest rhythm-maker.

Set *format_specification* to `'` or `'storage'`.

```
>>> print format(maker)
rhythmmakertools.RestRhythmMaker(
    forbidden_written_duration=durationtools.Duration(1, 4),
)
```

Returns string.

RestRhythmMaker.**__makenew__**(*args, **kwargs)

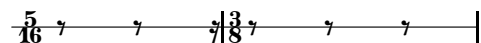
Creates new rest rhythm-maker with *kwargs*.

```
>>> new_maker = new(maker)
```

```
>>> print format(new_maker)
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, leaves)
```

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



Returns new rest rhythm-maker.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

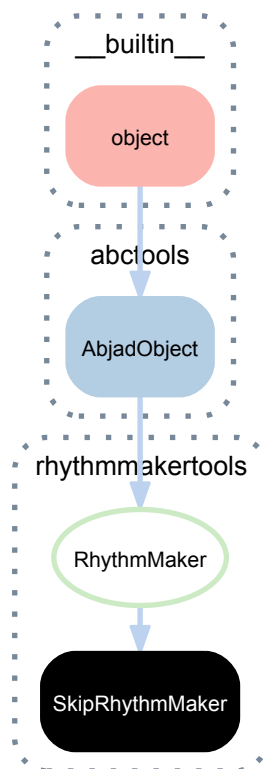
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.11 rhythmtools.SkipRhythmMaker



class `rhythmtools.SkipRhythmMaker`
Skip rhythm-maker:

```
>>> maker = rhythmtools.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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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

1 ——— | 3 ——— | 5 ——— |

1 16 8

Usage follows the two-step instantiate-then-call pattern shown here.

Bases

- `rhythmtools.RhythmMaker`
- `abctools.AbjadObject`
- `__builtin__.object`

Methods

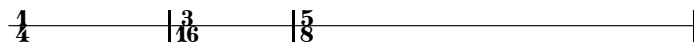
`SkipRhythmMaker.reverse()`
Reverses skip rhythm-maker.

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)
rhythmmakertools.SkipRhythmMaker()
```

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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



A musical staff with a single line. It contains three measures separated by bar lines. The first measure has a quarter note (1/4). The second measure has an eighth note (3/16). The third measure has a quarter note (5/8). The staff ends with a double bar line.

Returns new skip rhythm-maker.

Special methods

`SkipRhythmMaker.__call__(divisions, seeds=None)`
Calls 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.

`SkipRhythmMaker.__format__(format_specification='')`
Formats skip rhythm-maker.

Set *format_specification* to '' or 'storage'.

```
>>> print format(maker)
rhythmmakertools.SkipRhythmMaker()
```

Returns string.

`SkipRhythmMaker.__makenew__(*args, **kwargs)`
Creates new skip rhythm-maker with *kwargs*.

```
>>> new_maker = new(maker)
```

```
>>> print format(new_maker)
rhythmmakertools.SkipRhythmMaker()
```

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> leaf_lists = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(leaf_lists)
>>> measures = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = scoretools.RhythmicStaff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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

Returns new skip rhythm-maker.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

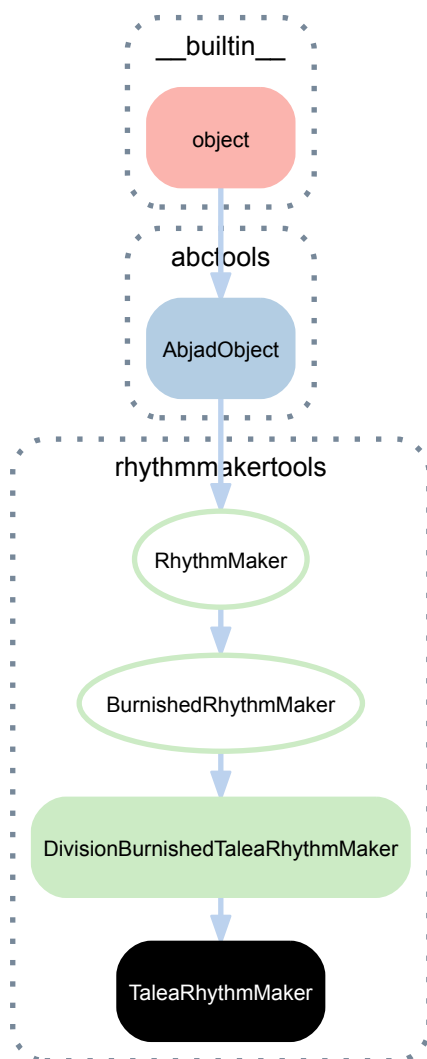
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.12 rhythm makertools.TaleaRhythmMaker



```
class rhythm makertools .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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.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.AbjadObject`
- `__builtin__.object`

Methods

`TaleaRhythmMaker.reverse()`
Reverses talea rhythm-maker.

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)
rhythm makertools.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 = \
...     scoretools.make_spacer_skip_measures(
...     divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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



Returns new talea rhythm-maker.

Special methods

(BurnishedRhythmMaker).**__call__**(*divisions*, *seeds=None*)

Calls 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.

TaleaRhythmMaker.**__format__**(*format_specification=''*)

Formats talea rhythm-maker.

Set *format_specification* to `'` or `'storage'`.

```
>>> print format(maker)
rhythm makertools.TaleaRhythmMaker(
  talea=[5],
  talea_denominator=16,
  prolation_addenda=[],
  secondary_divisions=[],
  beam_each_cell=False,
  beam_cells_together=False,
  tie_split_notes=True,
)
```

Returns string.

TaleaRhythmMaker.**__makenew__**(**args*, ***kwargs*)

Creates new talea rhythm-maker with *kwargs*.

```
>>> new_maker = new(maker, prolation_addenda=[1])
```

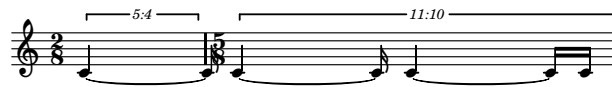
```
>>> print format(new_maker)
rhythm makertools.TaleaRhythmMaker(
  talea=[5],
```



```
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 = \
...     scoretools.make_spacer_skip_measures(
...         divisions)
>>> staff = Staff(measures)
>>> measures = scoretools.replace_contents_of_measures_in_expr(
...     staff, music)
```

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



Returns new talea rhythm-maker.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

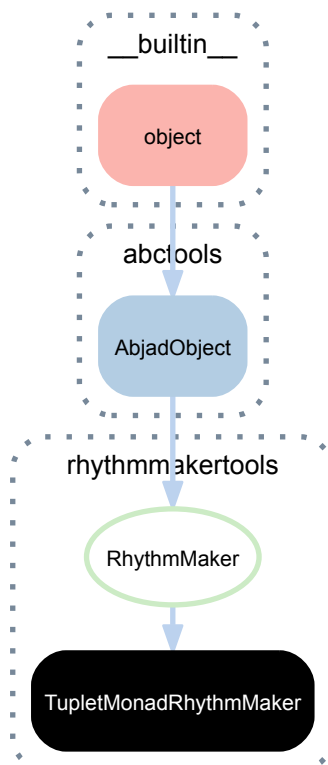
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2.13 rhythm makertools.TupletMonadRhythmMaker



`class` `rhythmmakertools.TupletMonadRhythmMaker` (*beam_each_cell=False*,
beam_cells_together=False)

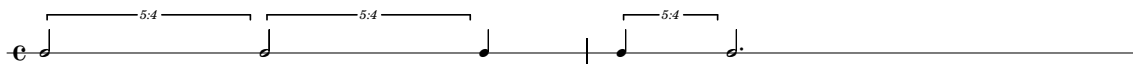
Tuplet monad rhythm-maker:

```
>>> 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 = scoretools.RhythmicStaff(tuplets)
```

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



Usage follows the two-step instantiate-then-call pattern shown here.

Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadObject`
- `__builtin__.object`

Methods

`TupletMonadRhythmMaker.reverse()`

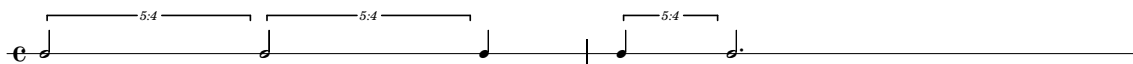
Reverses tuplet monad rhythm-maker.

```
>>> reversed_maker = maker.reverse()
```

```
>>> print format(reversed_maker)
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 = scoretools.RhythmicStaff(tuplets)
```

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



Returns new tuplet monad rhythm-maker.

Special methods

`TupletMonadRhythmMaker.__call__(divisions, seeds=None)`

Calls 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.

`TupletMonadRhythmMaker.__format__` (*format_specification*='')

Formats tuplet monad rhythm-maker.

Set *format_specification* to '' or 'storage'.

```
>>> print format(maker)
rhythmmakertools.TupletMonadRhythmMaker(
  beam_each_cell=False,
  beam_cells_together=False,
)
```

Returns string.

`TupletMonadRhythmMaker.__makenew__` (**args*, ***kwargs*)

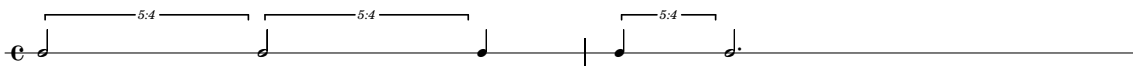
Creates new tuplet monad rhythm-maker with *kwargs*.

```
>>> new_maker = new(maker)
```

```
>>> print format(new_maker)
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 = scoretools.RhythmicStaff(tuplets)
```

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



Returns new tuplet monad rhythm-maker.

(`AbjadObject`).`__ne__` (*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

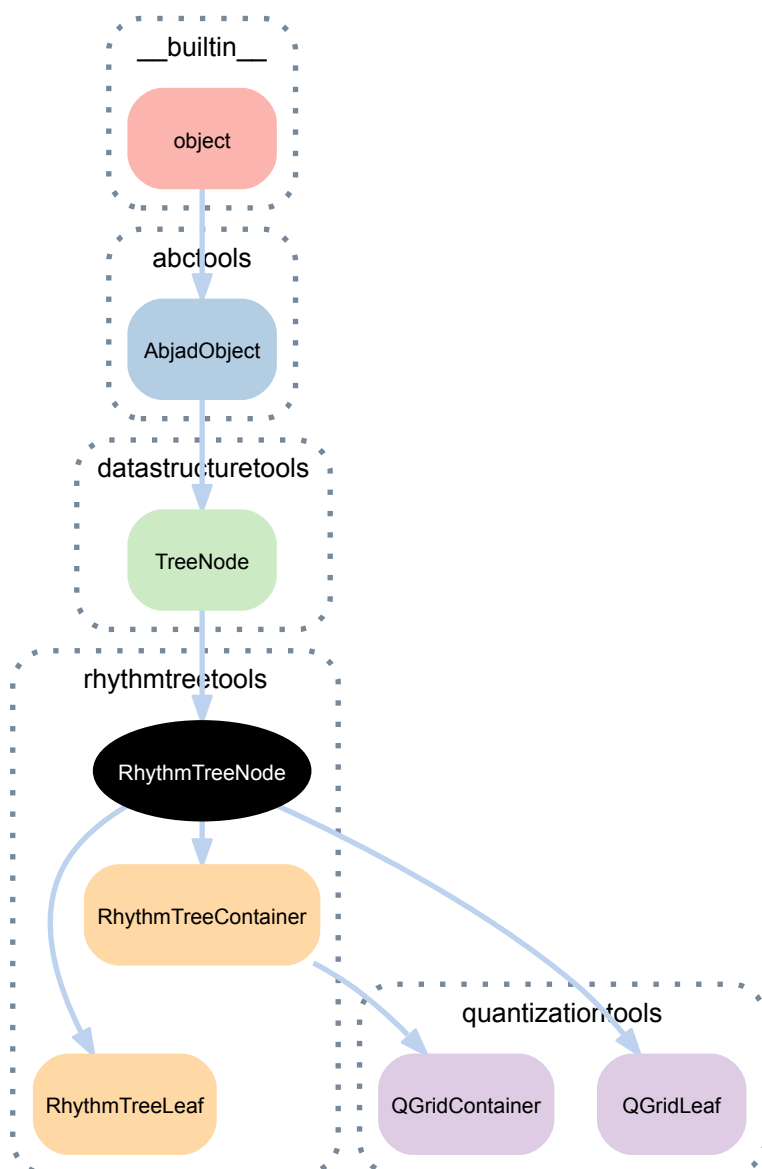
(`AbjadObject`).`__repr__` ()

Gets interpreter representation of Abjad object.

Returns string.

16.1 Abstract classes

16.1.1 `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 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**
Graph order of tree node.

Returns tuple.

RhythmTreeNode **.graphviz_format**
Graphviz format of rhythm tree node.

RhythmTreeNode **.graphviz_graph**
Graphviz graph of rhythm tree node.

(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 tree nodes.

(TreeNode) **.parent**
Parent of tree 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
```

Returns tree node.

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 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 = rhythmtreertools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
  (1 (
    1
  ))
  (1 (
    1
  )))
)
```

Returns string.

RhythmTreeNode.prolation

Prolation of rhythm tree node.

Returns multiplier.

RhythmTreeNode.prolations

Prolations of rhythm tree node.

Returns tuple.

(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 tree nodes.

(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
```

Returns tree node.

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**

Named of tree node.

Returns string.

`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)`

Calls rhythm tree node on *pulse_duration*.

`(TreeNode).__copy__(*args)`

Copies tree node.

Returns new tree node.

`(TreeNode).__deepcopy__(*args)`

Copies tree node.

Returns new tree node.

`(TreeNode).__eq__(expr)`

True when *expr* is a tree node. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(TreeNode).__ne__(expr)`

True when tree node does not equal *expr*. Otherwise false.

Returns boolean.

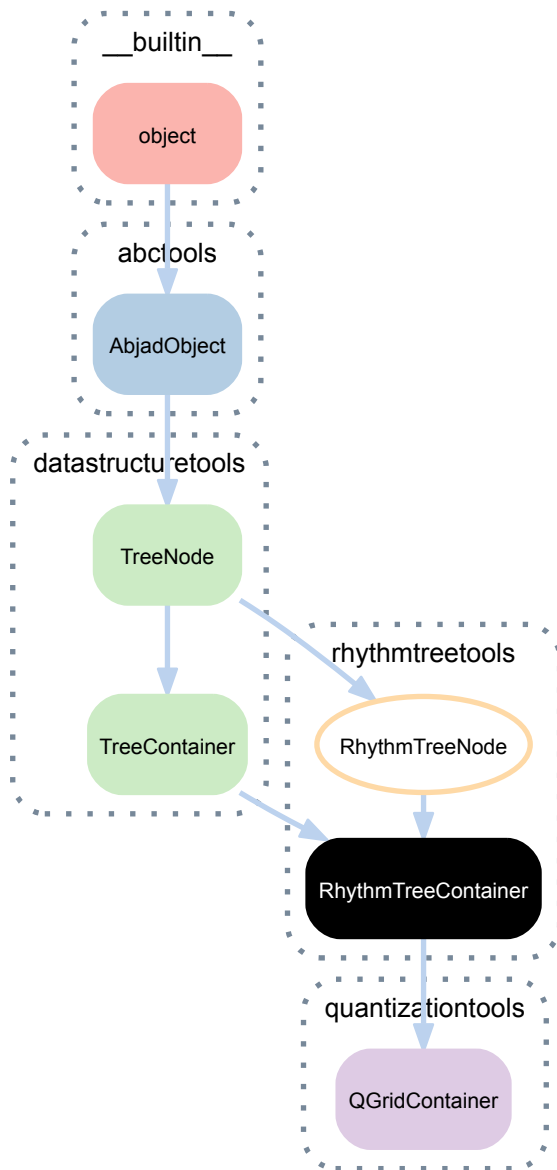
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

16.2 Concrete classes

16.2.1 `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 = rhythmtreertools.RhythmTreeContainer(
...     preprolated_duration=2)
>>> another_container.append(
...     rhythmtreertools.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(Duration(1, 4), "c'8 {@ 5:4 c'8., c'8 @}")]

```

Returns *RhythmTreeContainer* instance.

Bases

- `rhythmtreertools.RhythmTreeNode`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(TreeContainer).**children**
Children of tree container.

```
>>> 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 tree nodes.

(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 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**

Graph order of tree node.

Returns tuple.

(RhythmTreeNode).**graphviz_format**

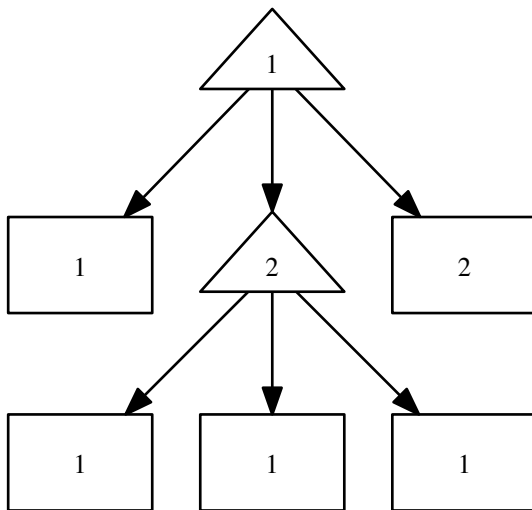
Graphviz format of rhythm tree node.

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;
}
```

```
>>> topleveltools.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 tree nodes.

(TreeContainer) **.leaves**

Leaves of tree container.

```
>>> 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 tree nodes produced by iterating tree container 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**

Parent of tree 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
```

Returns tree node.

(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 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**

Prolation of rhythm tree node.

Returns multiplier.

(RhythmTreeNode).**.prolations**

Prolations of rhythm tree node.

Returns tuple.

(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 tree nodes.

(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
```

Returns tree node.

`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`

Named of tree node.

Returns string.

`(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*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) . **extend** (*expr*)

Extendes *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) . **index** (*node*)

Indexes *node* in tree 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 tree 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)`
 Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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(Duration(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)
Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(*args)
Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
Deletes node *i* in tree container.

```
>>> 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.

(AbjadObject).**__format__**(*format_specification*='')
Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TreeContainer).__getitem__(i)`

Gets node *i* in tree 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.

`(TreeContainer).__iter__()`

Iterates tree container.

Yields children of tree container.

`(TreeContainer).__len__()`

Returns nonnegative integer number of nodes in container.

`(TreeNode).__ne__(expr)`

True when tree node does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

`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

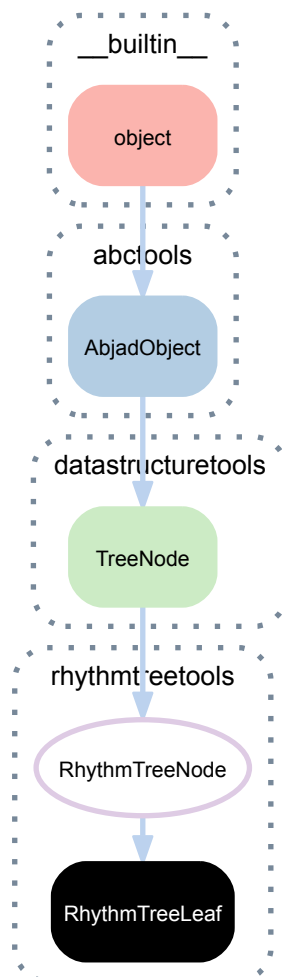
>>> c.parent is a
True

>>> b.parent is None
True

>>> a.children == (c,)
True
```

Return *None*.

16.2.2 `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..'),)
```

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 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**

Graph order of tree node.

Returns tuple.

(RhythmTreeNode).**graphviz_format**

Graphviz format of rhythm tree node.

RhythmTreeLeaf.**graphviz_graph**

Graphviz graph of rhythm tree leaf.

(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 tree nodes.

(TreeNode).parent

Parent of tree 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
```

Returns tree node.

(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 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**

Prolation of rhythm tree node.

Returns multiplier.

(RhythmTreeNode) **.prolations**

Prolations of rhythm tree node.

Returns tuple.

(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 tree nodes.

(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
```

Returns tree node.

RhythmTreeLeaf **.rtm_format**

RTM format of rhythm tree leaf.

```
>>> rhythmtreertools.RhythmTreeLeaf(1, is_pitched=True).rtm_format
'1'
>>> rhythmtreertools.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 = rhythmtreertools.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**

Gets and sets boolean equal to true if leaf is pitched. Otherwise false.

```
>>> leaf = rhythmtreetools.RhythmTreeLeaf()
>>> leaf.is_pitched
True
```

```
>>> leaf.is_pitched = False
>>> leaf.is_pitched
False
```

Returns boolean.

(TreeNode) **.name**

Named of tree node.

Returns string.

(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)

Copies tree node.

Returns new tree node.

(TreeNode) **.__deepcopy__** (*args)

Copies tree node.

Returns new tree node.

`RhythmTreeLeaf.__eq__(expr)`

True when *expr* is a rhythm tree leaf with preprolated duration and pitch boolean equal to those of this rhythm tree leaf. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TreeNode).__ne__(expr)`

True when tree node does not equal *expr*. Otherwise false.

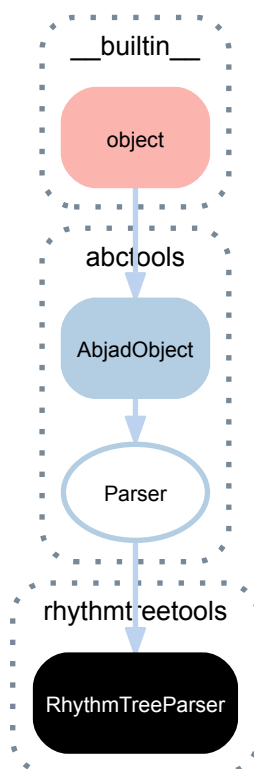
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

16.2.3 `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 LRParser 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*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(AbjadObject) `.__format__` (*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject) `.__ne__` (*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

16.3 Functions

16.3.1 `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(Duration(1, 4), "c'8 c'16 c'16 c'8")
```

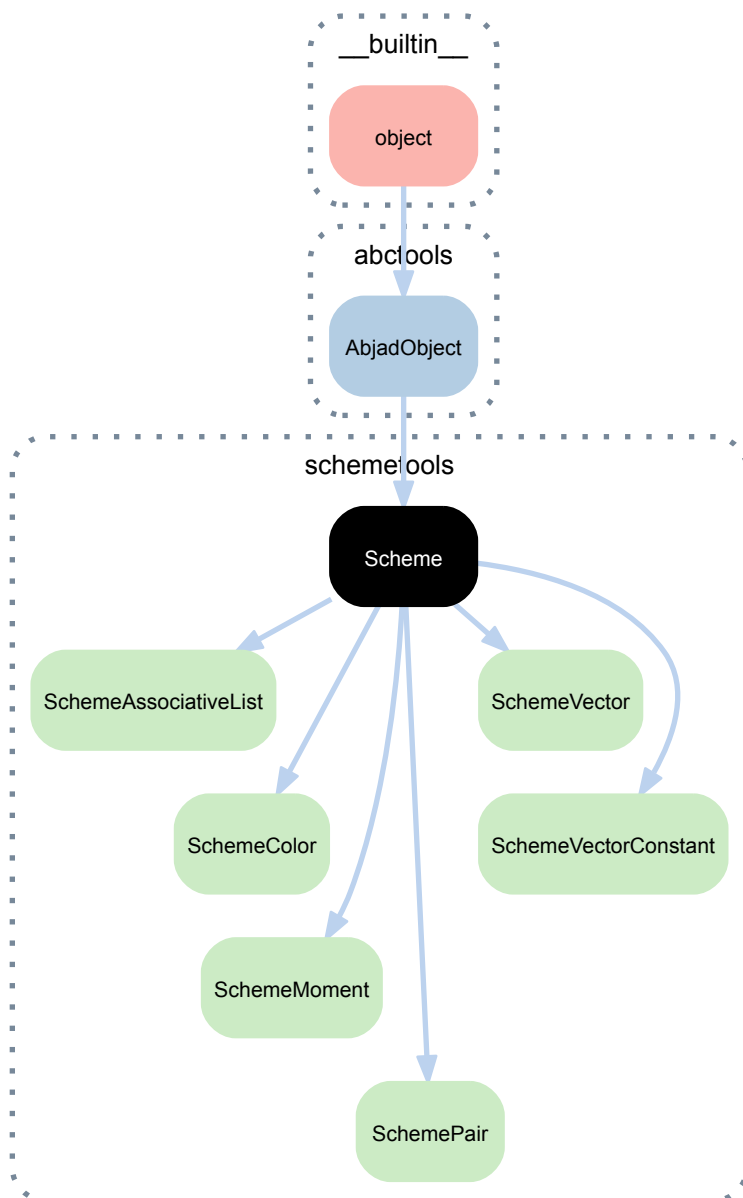
Also supports fractional durations:

```
>>> rtm = '(3/4 (1 1/2 (4/3 (1 -1/2 1))))'
>>> rhythmtreetools.parse_rtm_syntax(rtm)
FixedDurationTuplet(Duration(3, 16), "c'8 c'16 {@ 15:8 c'8, r16, c'8 @}")
>>> print format(_)
\tweak #'text #tuplet-number::calc-fraction-text
\times 9/17 {
  c'8
  c'16
  \times 8/15 {
    c'8
    r16
    c'8
  }
}
```

Returns fixed-duration tuplet or container.

17.1 Concrete classes

17.1.1 schemetools.Scheme



class `schemetools.Scheme` (**args, **kwargs*)
Abjad model of Scheme code.

```
>>> scheme = schemetools.Scheme(True)
>>> format(scheme)
'##t'
```

Scheme can represent nested structures:

```
>>> scheme = schemetools.Scheme(
...     ('left', (1, 2, False)), ('right', (1, 2, 3.3)))
>>> format(scheme)
'#((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))
>>> format(scheme_1) == format(scheme_2)
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="'#")
>>> format(scheme)
"'#(1 2 3)''
```

Scheme can also force quotes around strings which contain no whitespace:

```
>>> scheme = schemetools.Scheme('nospaces', force_quotes=True)
>>> print format(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`

True when quotes should be forced in output. Otherwise false.

Returns boolean.

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)`

True when *expr* is a scheme object with a value equal to that of this scheme object. Otherwise false.

Returns boolean.

`Scheme.__format__(format_specification='')`

Formats scheme.

Set *format_specification* to `''`, `'lilypond'` or `'storage'`. Interprets `'` equal to `'lilypond'`.

```
>>> scheme = schemetools.Scheme('foo')
>>> format(scheme)
'#foo'
```

```
>>> print format(scheme, 'storage')
schemetools.Scheme(
    'foo'
)
```

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

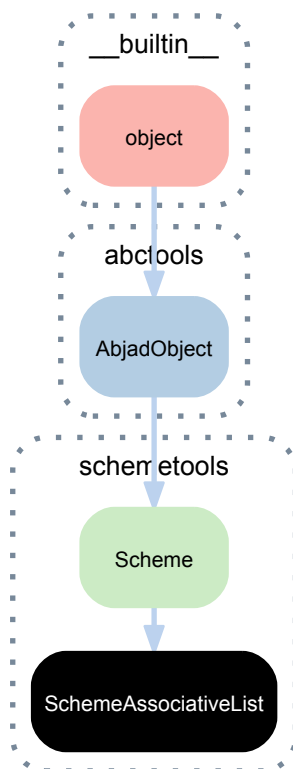
Returns string.

`Scheme.__str__()`

String representation of scheme object.

Returns string.

17.1.2 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**
 True when quotes should be forced in output. Otherwise false.
 Returns boolean.

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*)

True when *expr* is a scheme object with a value equal to that of this scheme object. Otherwise false.

Returns boolean.

(Scheme) .**__format__**(*format_specification*='')

Formats scheme.

Set *format_specification* to '', 'lilypond' or 'storage'. Interprets '' equal to 'lilypond'.

```
>>> scheme = schemetools.Scheme('foo')
>>> format(scheme)
'#foo'
```

```
>>> print format(scheme, 'storage')
schemetools.Scheme(
    'foo'
)
```

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

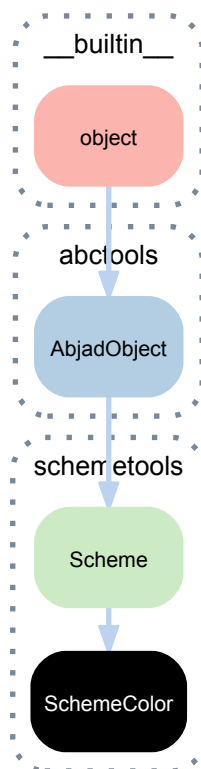
Returns string.

(Scheme) .**__str__**()

String representation of scheme object.

Returns string.

17.1.3 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`
True when quotes should be forced in output. Otherwise false.
Returns boolean.

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*)

True when *expr* is a scheme object with a value equal to that of this scheme object. Otherwise false.

Returns boolean.

(Scheme) .**__format__**(*format_specification*='')

Formats scheme.

Set *format_specification* to '', 'lilypond' or 'storage'. Interprets '' equal to 'lilypond'.

```
>>> scheme = schemetools.Scheme('foo')
>>> format(scheme)
'#foo'
```

```
>>> print format(scheme, 'storage')
schemetools.Scheme(
    'foo'
)
```

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

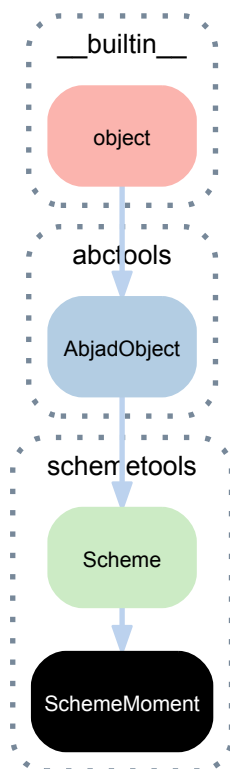
Returns string.

(Scheme) .**__str__**()

String representation of scheme object.

Returns string.

17.1.4 schemetools.SchemeMoment



class `schemetools.SchemeMoment` (*args, **kwargs)

A LilyPond scheme moment.

Initializes with two integers:

```
>>> moment = schemetools.SchemeMoment(1, 68)
>>> moment
SchemeMoment(1, 68)
```

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`

True when quotes should be forced in output. Otherwise false.

Returns boolean.

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)`

True when *arg* is a scheme moment with the same value as that of this scheme moment.

```
>>> moment == schemetools.SchemeMoment(1, 68)
True
```

Otherwise false.

```
>>> moment == schemetools.SchemeMoment(1, 54)
False
```

Returns boolean.

`(Scheme).__format__(format_specification='')`

Formats scheme.

Set *format_specification* to `""`, `'lilypond'` or `'storage'`. Interprets `""` equal to `'lilypond'`.

```
>>> scheme = schemetools.Scheme('foo')
>>> format(scheme)
'#foo'
```

```
>>> print format(scheme, 'storage')
schemetools.Scheme(
    'foo'
)
```

Returns string.

`SchemeMoment.__ge__(other)`

`x.__ge__(y) <==> x>=y`

`SchemeMoment.__gt__(other)`

`x.__gt__(y) <==> x>y`

`SchemeMoment.__le__(other)`

`x.__le__(y) <==> x<=y`

`SchemeMoment.__lt__(arg)`

True when *arg* is a scheme moment with value greater than that of this scheme moment.

```
>>> moment < schemetools.SchemeMoment(1, 32)
True
```

Otherwise false:

```
>>> moment < schemetools.SchemeMoment(1, 78)
False
```

Returns boolean.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

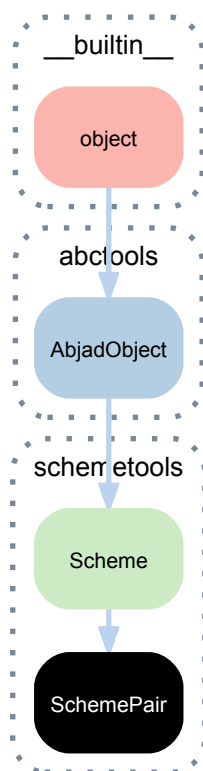
Returns string.

(Scheme).**__str__**()

String representation of scheme object.

Returns string.

17.1.5 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`

True when quotes should be forced in output. Otherwise false.

Returns boolean.

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)`

True when *expr* is a scheme object with a value equal to that of this scheme object. Otherwise false.

Returns boolean.

`SchemePair.__format__(format_specification='')`

Formats Scheme pair.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`. Interprets `'` equal to `'lilypond'`.

```
>>> scheme_pair = schemetools.SchemePair(-1, 1)
```

```
>>> format(scheme_pair)
"#' (-1 . 1)"
```

```
>>> print format(scheme_pair, 'storage')
schemetools.SchemePair(
    -1,
    1
)
```

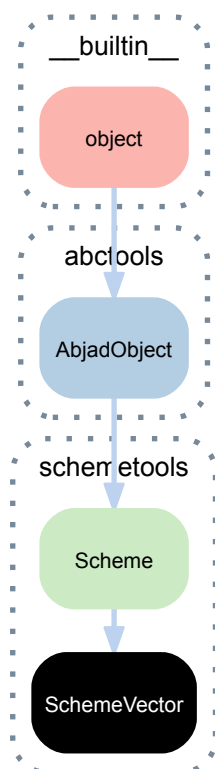
Returns string.

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

(Scheme).**__str__**()
 String representation of scheme object.
 Returns string.

17.1.6 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**

True when quotes should be forced in output. Otherwise false.

Returns boolean.

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)

True when *expr* is a scheme object with a value equal to that of this scheme object. Otherwise false.

Returns boolean.

(Scheme) .**__format__** (format_specification='')

Formats scheme.

Set *format_specification* to *''*, *'lilypond'* or *'storage'*. Interprets *''* equal to *'lilypond'*.

```
>>> scheme = schemetools.Scheme('foo')
>>> format(scheme)
'#foo'
```

```
>>> print format(scheme, 'storage')
schemetools.Scheme(
    'foo'
)
```

Returns string.

(AbjadObject) .**__ne__** (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

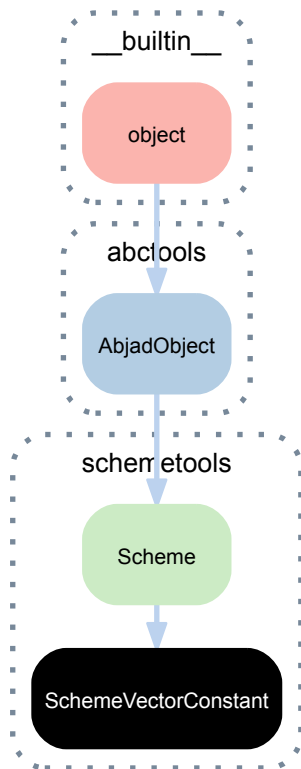
(AbjadObject) .**__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

`(Scheme).__str__()`
 String representation of scheme object.
 Returns string.

17.1.7 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`
 True when quotes should be forced in output. Otherwise false.
 Returns boolean.

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)
True when *expr* is a scheme object with a value equal to that of this scheme object. Otherwise false.
Returns boolean.

(Scheme) .**__format__** (format_specification='')
Formats scheme.
Set *format_specification* to '', 'lilypond' or 'storage'. Interprets '' equal to 'lilypond'.

```
>>> scheme = schemetools.Scheme('foo')
>>> format(scheme)
'#foo'
```

```
>>> print format(scheme, 'storage')
schemetools.Scheme(
    'foo'
)
```

Returns string.

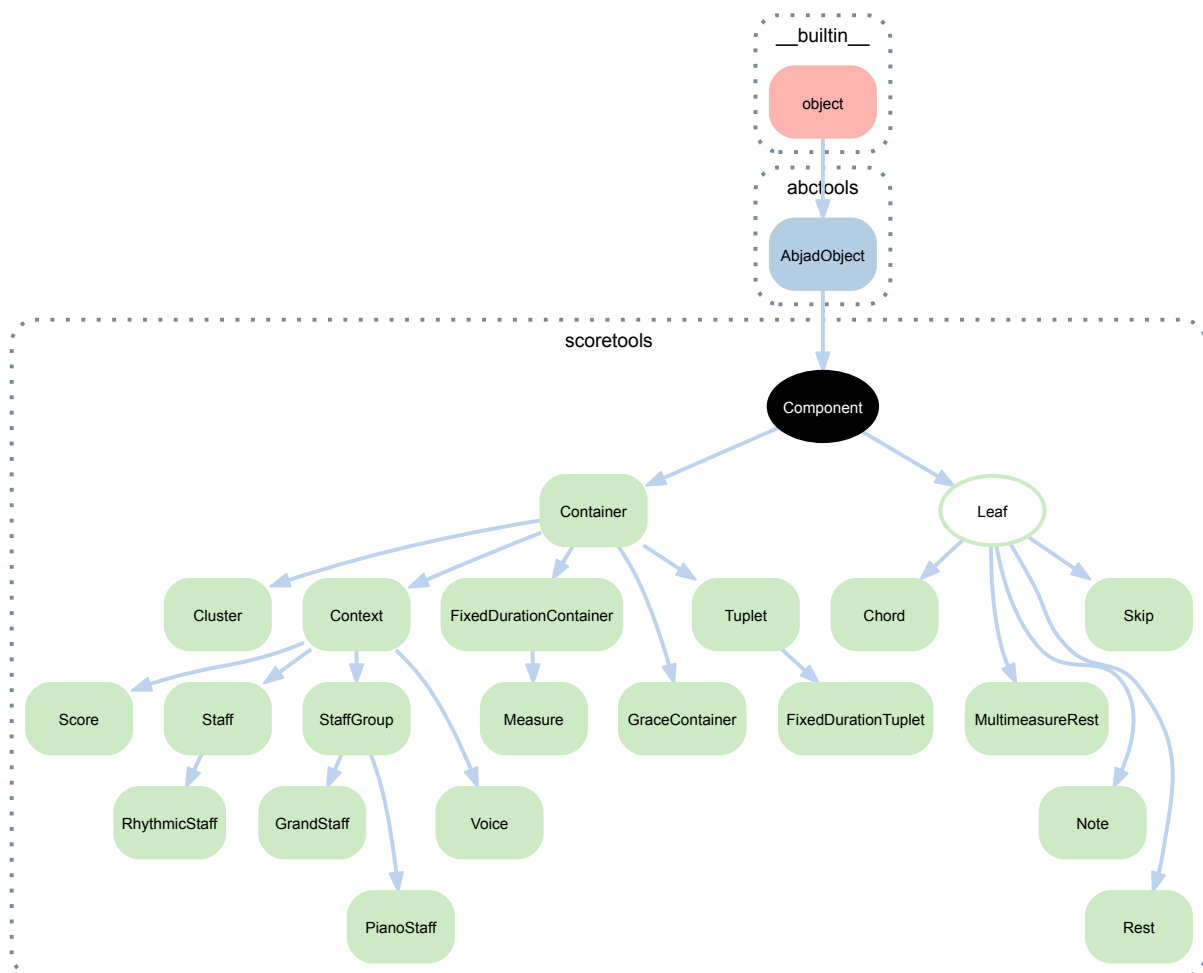
(AbjadObject) .**__ne__** (expr)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject) .**__repr__** ()
Gets interpreter representation of Abjad object.
Returns string.

(Scheme) .**__str__** ()
String representation of scheme object.
Returns string.

18.1 Abstract classes

18.1.1 scoretools.Component



class `scoretools.Component`

A score component.

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

Bases

- `abctools.AbjadObject`

- `__builtin__.object`

Special methods

`Component.__copy__(*args)`

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

Returns new component.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`Component.__format__(format_specification='')`

Formats component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

`Component.__illustrate__()`

Illustrates component.

Returns LilyPond file.

`Component.__mul__(n)`

Copies component *n* times and detaches spanners.

Returns list of new components.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`Component.__repr__()`

Gets interpreter representation of leaf.

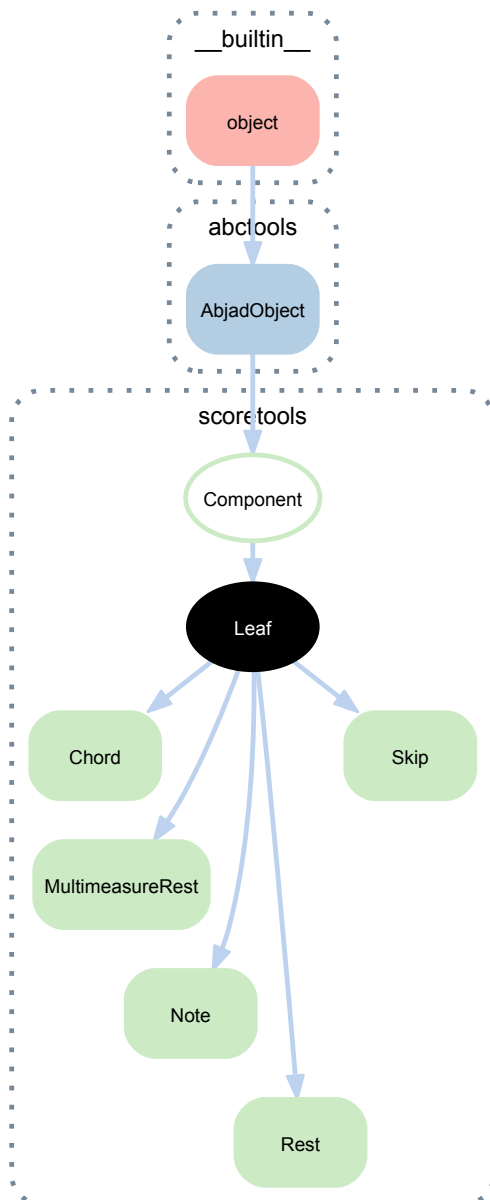
Returns string.

`Component.__rmul__(n)`

Copies component *n* times and detach spanners.

Returns list of new components.

18.1.2 scoretools.Leaf



class `scoretools.Leaf` (*written_duration*)

Abstract base class from which notes, rests, chords and skips inherit.

Bases

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

Read/write properties

`Leaf.written_duration`

Written duration of leaf.

Set to duration.

Returns duration.

Special methods

(Component).**__copy__**(*args)

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

Returns new component.

(AbjadObject).**__eq__**(expr)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(format_specification='')

Formats component.

Set *format_specification* to '', 'lilypond' or 'storage'.

Returns string.

(Component).**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(Component).**__mul__**(n)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject).**__ne__**(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Component).**__repr__**()

Gets interpreter representation of leaf.

Returns string.

(Component).**__rmul__**(n)

Copies component *n* times and detach spanners.

Returns list of new components.

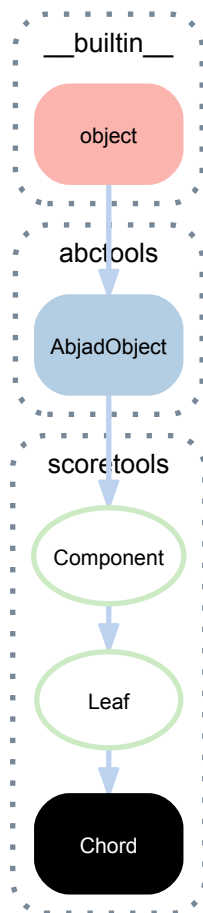
Leaf.**__str__**()

String representation of leaf.

Returns string.

18.2 Concrete classes

18.2.1 scoretools.Chord



```
class scoretools.Chord(*args)
    A chord.
```

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



Bases

- `scoretools.Leaf`
- `scoretools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

Read/write properties

`Chord.note_heads`
Note heads in chord.

Example 1. Get note heads in chord:

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



```
>>> print format(chord.note_heads)
scoretools.NoteHeadInventory(
  [
    scoretools.NoteHead(
      written_pitch=pitchtools.NamedPitch("g'"),
      is_cautionary=False,
      is_forced=False,
    ),
    scoretools.NoteHead(
      written_pitch=pitchtools.NamedPitch("c''"),
      is_cautionary=False,
      is_forced=False,
    ),
    scoretools.NoteHead(
      written_pitch=pitchtools.NamedPitch("e''"),
      is_cautionary=False,
      is_forced=False,
    ),
  ]
)
```

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.

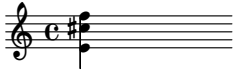
Returns tuple.

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.

`Chord.written_pitches`

Written pitches in chord.

Example 1. Get written pitches:

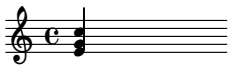
```
>>> chord = Chord("<g' c' e''>4")
>>> show(chord)
```



```
>>> 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.

Special methods

`(Component).__copy__(*args)`

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

Returns new component.

(AbjadObject) .**__eq__**(*expr*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

(Component) .**__format__**(*format_specification*='')
Formats component.
Set *format_specification* to '', 'lilypond' or 'storage'.
Returns string.

(Component) .**__illustrate__**()
Illustrates component.
Returns LilyPond file.

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

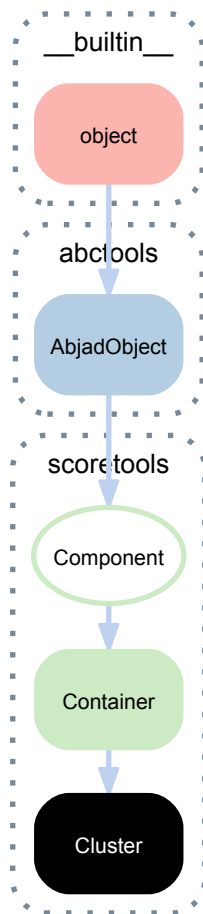
(AbjadObject) .**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(Component) .**__repr__**()
Gets 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.

18.2.2 scoretools.Cluster



class `scoretools.Cluster` (*music=None*)
A cluster.

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



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

Bases

- `scoretools.Container`
- `scoretools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

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)
```

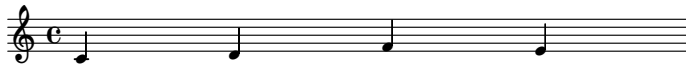


Returns none.

(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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

(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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification=''*)

Formats component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

(Container).**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component).**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

Cluster.**__repr__**()

Gets interpreter representation of cluster.

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

Returns string.

(Component).**__rmul__**(*n*)

Copies component *n* times and detach spanners.

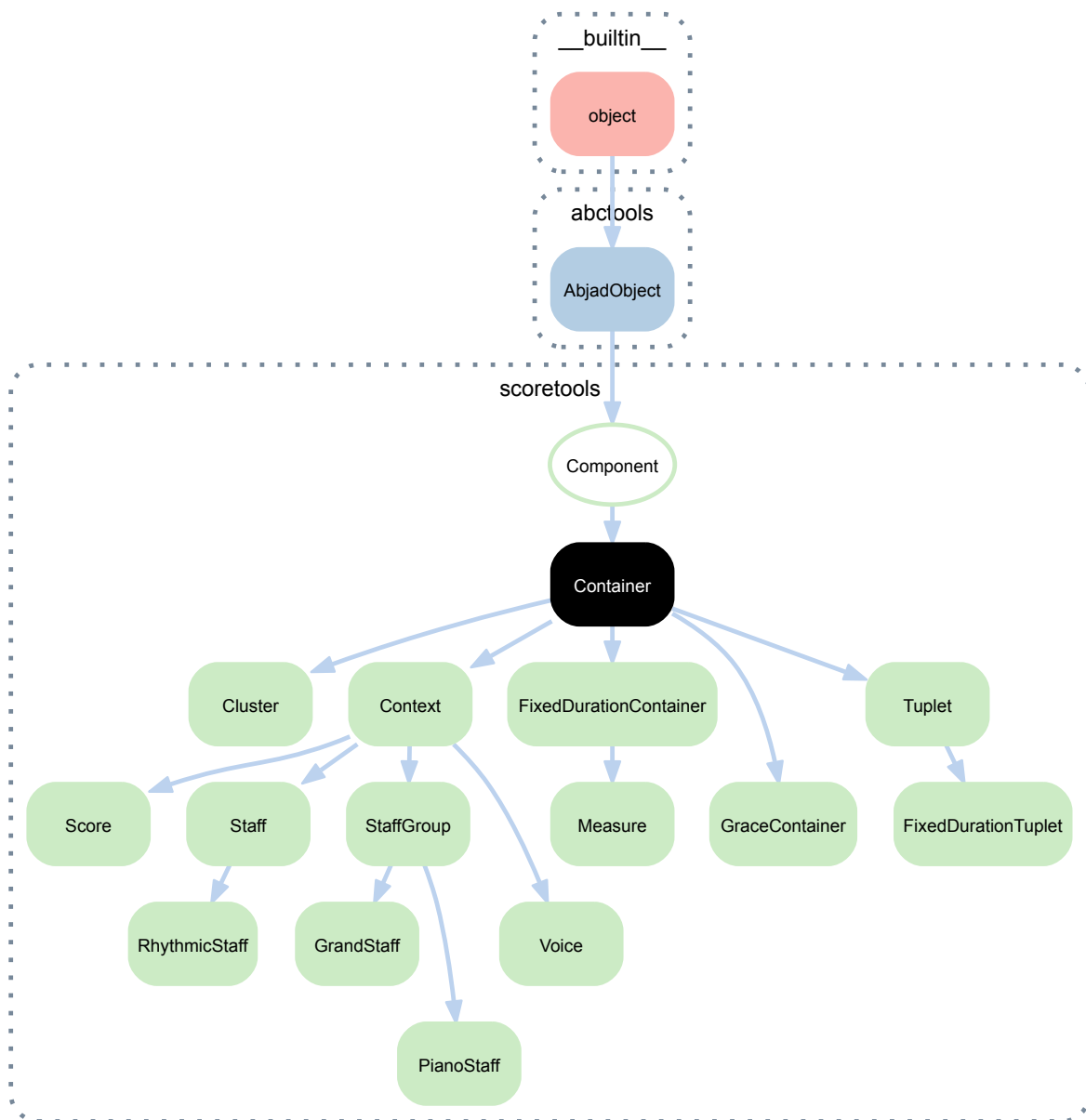
Returns list of new components.

(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.

18.2.3 scoretools.Container

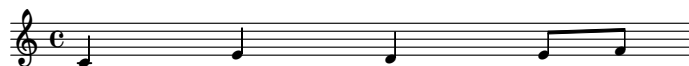


class `scoretools.Container` (*music=None*)

An iterable container of music.

Example:

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



Bases

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

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)
```

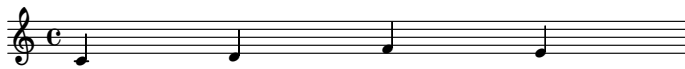


Returns none.

`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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, container[:])
>>> slur.direction = Down
>>> show(container)
```



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



Returns none.

`Container.pop(i=-1)`

Removes 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.

`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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(Component).__format__` (*format_specification=''*)

Formats component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

`Container.__getitem__` (*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

`(Component).__illustrate__` ()

Illustrates component.

Returns LilyPond file.

`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)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(Component).__repr__()`

Gets interpreter representation of leaf.

Returns string.

`(Component).__rmul__(n)`

Copies component *n* times and detach spanners.

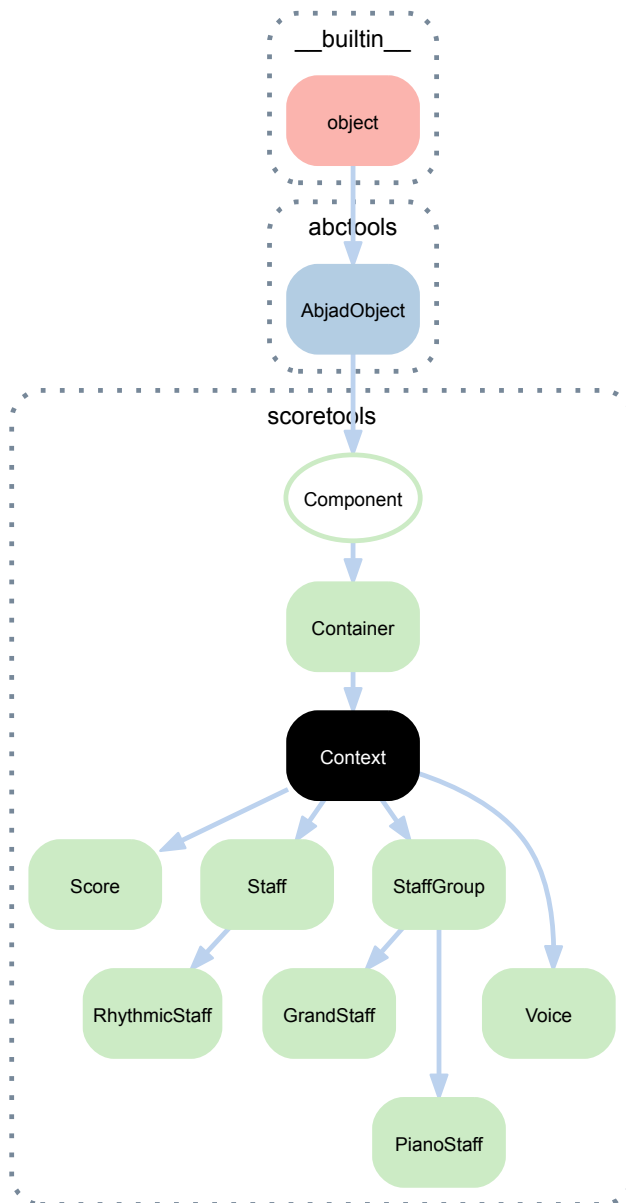
Returns list of new components.

`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.

18.2.4 scoretools.Context



class `scoretools.Context` (*music=None*, *context_name='Context'*, *name=None*)
 A horizontal layer of music.

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

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

Bases

- `scoretools.Container`
- `scoretools.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')
>>> print format(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')
>>> print format(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

Context.**is_semantic**

True when context is semantic. Otherwise false.

Returns boolean.

Read/write properties

Context.**context_name**

Gets and sets context name of context.

Returns string.

Context.**is_nonsemantic**

Gets and sets nonsemantic voice flag.

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

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

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

Gets nonsemantic voice 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**

Gets and sets name of context.

Returns 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)
```

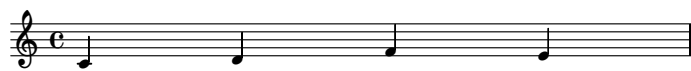


Returns none.

(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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

(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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification=''*)

Formats component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

(Container).**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component).**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

Context .**__repr__**()
 Gets interpreter representation of context.

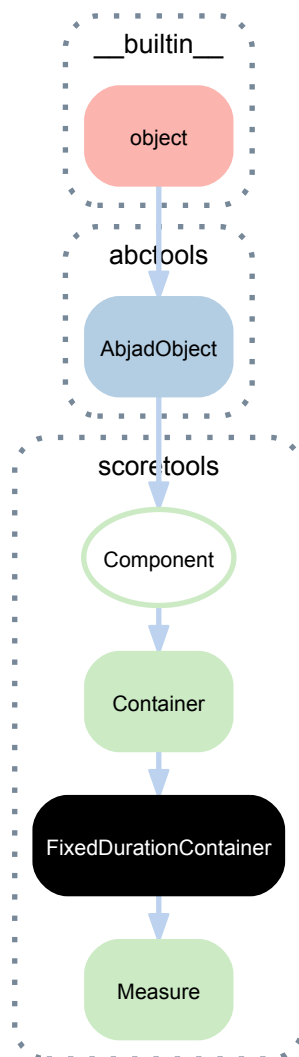
```
>>> context
TimeSignatureContext-MeterVoice{}
```

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.

18.2.5 scoretools.FixedDurationContainer



class `scoretools.FixedDurationContainer` (*target_duration=None, music=None, **kwargs*)
 A fixed-duration container.

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



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

Returns fixed-duration container.

Bases

- `scoretools.Container`
- `scoretools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`FixedDurationContainer.is_full`

True when preprolated duration equals target duration. Otherwise false.

Returns boolean.

`FixedDurationContainer.is_misfilled`

True when preprolated duration does not equal target duration. Otherwise false.

Returns boolean.

`FixedDurationContainer.is_overfull`

True when preprolated duration is greater than target duration. Otherwise false.

Returns boolean.

`FixedDurationContainer.is_underfull`

True when preprolated duration is less than target duration. Otherwise false.

Returns boolean.

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`

Gets and sets target duration of fixed-duration container.

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)
```



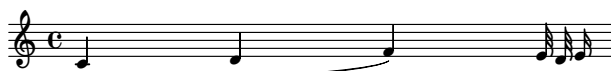
Returns none.

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

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



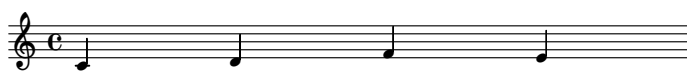
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
```



```
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

(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 indicators 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*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(Component) .**__format__**(*format_specification*='')
 Formats component.
 Set *format_specification* to '', 'lilypond' or 'storage'.
 Returns string.

(Container) .**__getitem__**(*i*)
 Gets container *i*.
 Traverses top-level items only.
 Returns component.

(Component) .**__illustrate__**()
 Illustrates component.
 Returns LilyPond file.

(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.

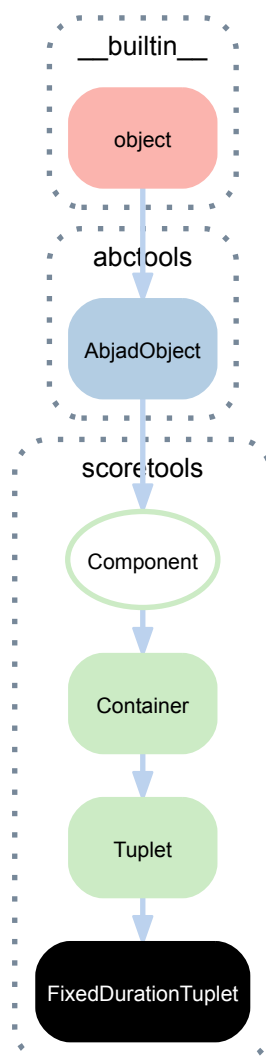
(AbjadObject) .**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

FixedDurationContainer .**__repr__**()
 Gets interpreter representation of fixed-duration container.
 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.

18.2.6 scoretools.FixedDurationTuplet



class `scoretools.FixedDurationTuplet` (*duration*, *music=None*)
 A tuplet with fixed duration and variable multiplier.

```
>>> tuplet = scoretools.FixedDurationTuplet(Duration(2, 8), [])
>>> tuplet.extend("c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.append("fs'4")
>>> show(tuplet)
```



Bases

- `scoretools.Tuplet`
- `scoretools.Container`
- `scoretools.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.

`FixedDurationTuplet.multiplied_duration`

Multiplied duration of tuplet:

```
>>> tuplet = scoretools.FixedDurationTuplet((1, 4), "c'8 d'8 e'8")
>>> tuplet.multiplied_duration
Duration(1, 4)
```

Returns duration.

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 1. Get tuplet invisibility status:

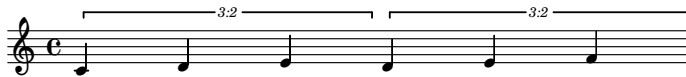
```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



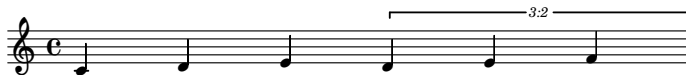
```
>>> tuplet.is_invisible is None
True
```

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 = scoretools.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`

Gets and sets target duration of fixed-duration tuplet.

```
>>> tuplet = scoretools.FixedDurationTuplet(
...     (1, 4), "c'8 d'8 e'8")
>>> tuplet.target_duration
Duration(1, 4)
```



```
>>> tuplet.target_duration = Duration(5, 8)
>>> print format(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)
```

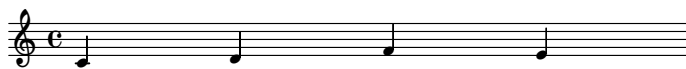


Returns none.

(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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

`(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*.

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.

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet
Tuplet(Multiplier(2, 3), "c'8 d'8 e'8")
```

```
>>> new_tuplet = tuplet.to_fixed_duration_tuplet()
>>> show(new_tuplet)
```



```
>>> new_tuplet
FixedDurationTuplet(Duration(1, 4), "c'8 d'8 e'8")
```

Returns new tuplet.

FixedDurationTuplet.**to_fixed_multiplier**()
Change fixed-duration tuplet to (unqualified) tuplet.

```
>>> tuplet = scoretools.FixedDurationTuplet((2, 8), [])
>>> tuplet.extend("c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet
FixedDurationTuplet(Duration(1, 4), "c'8 d'8 e'8")
```

```
>>> new_tuplet = tuplet.to_fixed_multiplier()
>>> show(new_tuplet)
```



```
>>> new_tuplet
Tuplet(Multiplier(2, 3), "c'8 d'8 e'8")
```

Returns new tuplet.

FixedDurationTuplet.**toggle_prolation**()
Toggles prolation of fixed-duration tuplet.

```
>>> tuplet = scoretools.FixedDurationTuplet((1, 4), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.toggle_prolation()
>>> show(tuplet)
```



```
>>> tuplet.toggle_prolation()
>>> show(tuplet)
```



Returns none.

FixedDurationTuplet.**trim**(*start*, *stop*='unused')

Trim fixed-duration tuplet elements from *start* to *stop*:

```
>>> tuplet = scoretools.FixedDurationTuplet(
...     Duration(2, 8), "c'8 d'8 e'8")
>>> tuplet
FixedDurationTuplet(Duration(1, 4), "c'8 d'8 e'8")
```

```
>>> tuplet.trim(2)
>>> tuplet
FixedDurationTuplet(Duration(1, 6), "c'8 d'8")
```

Preserve fixed-duration tuplet multiplier.

Adjust fixed-duration tuplet duration.

Returns none.

Static methods

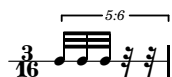
(Tuplet).**from_duration_and_ratio**(*duration*, *ratio*, *avoid_dots*=True, *decrease_durations_monotonically*=True, *is_diminution*=True)

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:

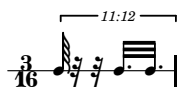
```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=True,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = scoretools.RhythmicStaff([measure])
>>> show(staff)
```



Allow tupletted leaves to return with dots when some *ratio* do not equal 1:

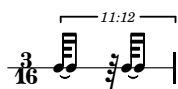
```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, -2, -2, 3, 3),
...     avoid_dots=True,
...     is_diminution=False,
```

```
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = scoretools.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 = scoretools.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 = scoretools.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 = scoretools.RhythmicStaff([measure])
>>> show(staff)
```

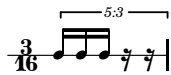


Example 3. Make diminished tuplet from *duration* and nonzero integer *ratio*.

Make tupletted leaves strictly without dots when all *ratio* equal 1:

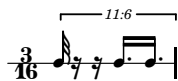
```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(1, 1, 1, -1, -1),
...     avoid_dots=True,
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
```

```
>>> staff = scoretools.RhythmicStaff([measure])
>>> show(staff)
```



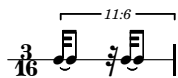
Allow tupletted leaves to return with dots when some *ratio* do not equal 1:

```
>>> 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 = scoretools.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 = scoretools.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 = scoretools.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 = scoretools.RhythmicStaff([measure])
>>> show(measure)
```



Reduces *ratio* relative to each other.

Interprets negative *ratio* as rests.

Returns fixed-duration tuplet.

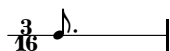
(Tuplet).**from_leaf_and_ratio**(*leaf*, *ratio*, *is_diminution*=True)

Makes tuplet from *leaf* and *ratio*.

```
>>> note = Note("c'8.")
```

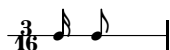
Example 1a. Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.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(scoretools.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(scoretools.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(scoretools.RhythmicStaff([measure]))
```

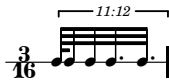


Example 1e. Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3, 3],
...     is_diminution=False,
... )
```



```
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```



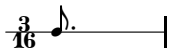
Example 1f. Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2, 3, 3, 4),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```



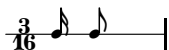
Example 2a. Change leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```



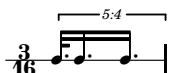
Example 2b. Change leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```



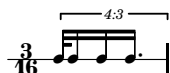
Example 2c. Change leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```

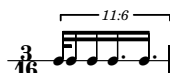


Example 2d. Change leaf to diminished tuplets with *ratio*:

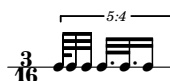
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```

**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(scoretools.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(scoretools.RhythmicStaff([measure]))
```



Returns tuplet.

(Tuplet).**from_nonreduced_ratio_and_nonreduced_fraction**(*ratio*, *fraction*)

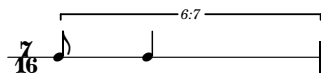
Makes tuplet from nonreduced *ratio* and nonreduced *fraction*.

Example 1. Make container when no prolotion is necessary:

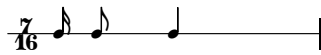
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = scoretools.RhythmicStaff([measure])
>>> show(staff)
```

**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 = scoretools.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 = scoretools.RhythmicStaff([measure])
>>> show(staff)
```



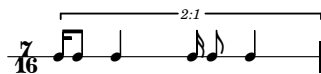
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1),
...     mathtools.NonreducedFraction(7, 16),
...     )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = scoretools.RhythmicStaff([measure])
>>> show(staff)
```



```
>>> 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 = scoretools.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 = scoretools.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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification*='')

Formats component.

Set *format_specification* to '', 'lilypond' or 'storage'.

Returns string.

(Container) .**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component) .**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

FixedDurationTuplet .**__repr__**()

Gets interpreter representation of fixed-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.

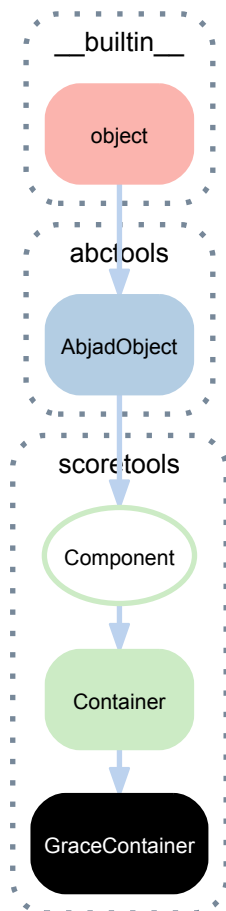
Returns none.

FixedDurationTuplet .**__str__**()

String representation of fixed-duration tuplet.

Returns string.

18.2.7 scoretools.GraceContainer



class `scoretools.GraceContainer` (*music=None, kind='grace'*)
 A container of grace music.

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



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



```
>>> notes = [Note("e'16"), Note("f'16")]
>>> after_grace = scoretools.GraceContainer(notes, kind='after')
>>> attach(after_grace, voice[1])
>>> show(voice)
```



Fill grace containers with notes, rests or chords.

Attach grace containers to nongrace notes, rests or chords.

Bases

- `scoretools.Container`
- `scoretools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

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")
>>> note = Note("cs'16")
>>> grace = scoretools.GraceContainer([note], kind='grace')
>>> attach(grace, staff[1])
>>> grace.kind
'grace'
```

Sets *kind* of grace container:

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

```
>>> grace.kind
'acciaccatura'
```

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

Returns string.

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)
```



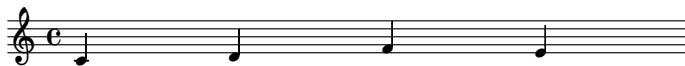
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

(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 indicators 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)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(Component) .**__format__** (format_specification='')
 Formats component.
 Set *format_specification* to '', 'lilypond' or 'storage'.
 Returns string.

(Container) .**__getitem__** (i)
 Gets container *i*.
 Traverses top-level items only.
 Returns component.

(Component) .**__illustrate__** ()
 Illustrates component.
 Returns LilyPond file.

(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.

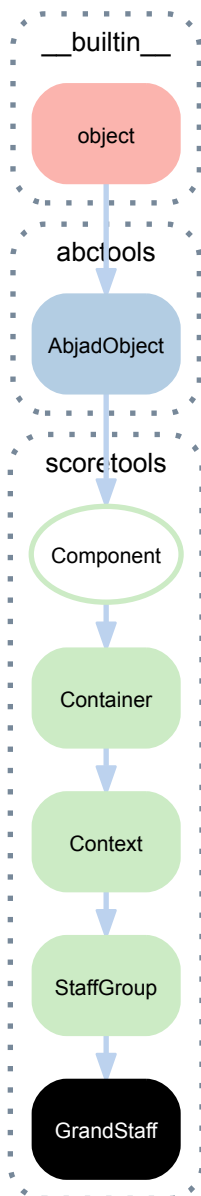
(AbjadObject) .**__ne__** (expr)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

GraceContainer .**__repr__** ()
 Gets interpreter representation of grace container.
 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.

18.2.8 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`
- `scoretools.Context`
- `scoretools.Container`
- `scoretools.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')
>>> print format(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')
>>> print format(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

`(Context).is_semantic`

True when context is semantic. Otherwise false.

Returns boolean.

Read/write properties

`(Context).context_name`

Gets and sets context name of context.

Returns string.

`(Context).is_nonsemantic`

Gets and sets nonsemantic voice flag.

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

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

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

Gets nonsemantic voice 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**

Gets and sets name of context.

Returns 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)
```



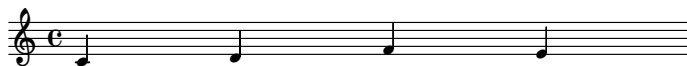
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, container[:])
>>> slur.direction = Down
>>> show(container)
```



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

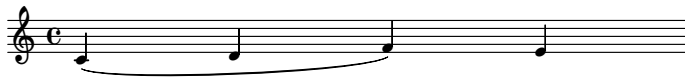


Returns none.

(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.

(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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification=''*)

Formats component.

Set *format_specification* to *'*, *'lilypond'* or *'storage'*.

Returns string.

(Container).**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component) .**__illustrate__**()
Illustrates component.

Returns LilyPond file.

(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*)
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Context) .**__repr__**()
Gets interpreter representation of context.

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

Returns string.

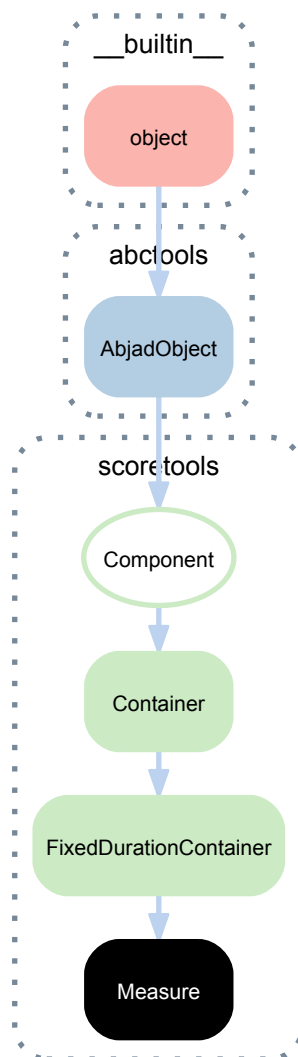
(Component) .**__rmul__**(*n*)
Copies component *n* times and detach spanners.

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.

18.2.9 scoretools.Measure



class scoretools.**Measure** (*time_signature=None, music=None*)
 A measure.

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
>>> show(measure)
```



Bases

- scoretools.FixedDurationContainer
- scoretools.Container
- scoretools.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'")
>>> show(measure)
```



```
>>> measure.has_non_power_of_two_denominator
True
```

Otherwise false:

```
>>> measure = Measure((5, 8), "c'8 d' e' f' g'")
>>> show(measure)
```



```
>>> 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'")
>>> show(measure)
```



```
>>> measure.has_power_of_two_denominator
True
```

Otherwise false:

```
>>> measure = Measure((5, 9), "c'8 d' e' f' g'")
>>> show(measure)
```



```
>>> 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'")
>>> show(measure)
```



```
>>> measure.implied_prolation
Multiplier(2, 3)
```

Returns multiplier.

Measure.is_full

True when measure duration equals time signature duration.

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
>>> show(measure)
```



```
>>> measure.is_full
True
```

Otherwise false.

Returns boolean.

Measure.is_misfilled

True when measure is either underfull or overfull.

```
>>> 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'")
>>> show(measure)
```



```
>>> measure.is_misfilled
False
```

Returns boolean.

Measure.is_overfull

True when measure 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 measure duration is less than time signature duration.

```
>>> measure = Measure((3, 4), "c'4 d'")
```

```
>>> measure.is_underfull
True
```

Otherwise false.

Returns boolean.

Measure.measure_number

1-indexed measure number.

```
>>> staff = Staff()
>>> staff.append(Measure((3, 4), "c' d' e'"))
>>> staff.append(Measure((2, 4), "f' g'"))
>>> show(staff)
```



```
>>> staff[0].measure_number
1
```

```
>>> staff[1].measure_number
2
```

Returns positive integer.

Measure.target_duration

Target duration of measure is always equal to duration of effective time signature.

```
>>> measure = Measure((3, 4), "c'4 d'4 e'4")
>>> measure.target_duration
Duration(3, 4)
```

Returns duration.

Measure.time_signature

Effective time signature of measure.

```
>>> measure.time_signature
TimeSignature((3, 4))
```

Returns time signature or none.

Read/write properties

Measure.always_format_time_signature

Gets and sets flag to indicate whether time signature should appear in LilyPond format even when not expected.

```
>>> measure.always_format_time_signature
False
```

Set to true when necessary to print the same signature repeatedly.

Defaults to false.

Returns boolean.

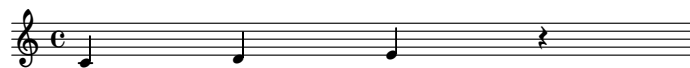
Measure.automatically_adjust_time_signature

Gets and sets flag to indicate whether time signature should update automatically following mutation.

```
>>> measure = Measure((3, 4), "c' d' e'")
>>> show(measure)
```



```
>>> measure.automatically_adjust_time_signature = True
>>> measure.append('r')
>>> show(measure)
```



Defaults 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)
```

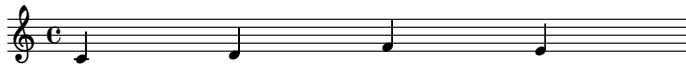


Returns none.

(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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

(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)

Adds two measures together in-score or outside-of-score.

Interface to `scoretools.fuse_measures()`.

(Container).**__contains__**(expr)

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

Returns boolean.

(Component).**__copy__**(*args)

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

Returns new component.

Measure.**__delitem__**(i)

Deletes *i* from measure.

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
>>> measure.automatically_adjust_time_signature = True
>>> show(measure)
```



```
>>> del(measure[1])
>>> show(measure)
```



Returns none.

(AbjadObject) .**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component) .**__format__**(*format_specification*='')

Formats component.

Set *format_specification* to ' ', 'lilypond' or 'storage'.

Returns string.

(Container) .**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component) .**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

Measure .**__repr__**()

Gets interpreter representation of measure.

```
>>> measure = Measure((3, 8), "c'8 d'8 e'8")
>>> show(measure)
```



```
>>> measure
Measure((3, 8), "c'8 d'8 e'8")
```

Returns string.

(Component) .**__rmul__**(*n*)

Copies component *n* times and detach spanners.

Returns list of new components.

Measure .**__setitem__**(*i*, *expr*)

Sets container item *i* to *expr*.

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
>>> measure.automatically_adjust_time_signature = True
>>> show(measure)
```



```
>>> measure[1] = Note("ds'8.")
>>> show(measure)
```



Returns none.

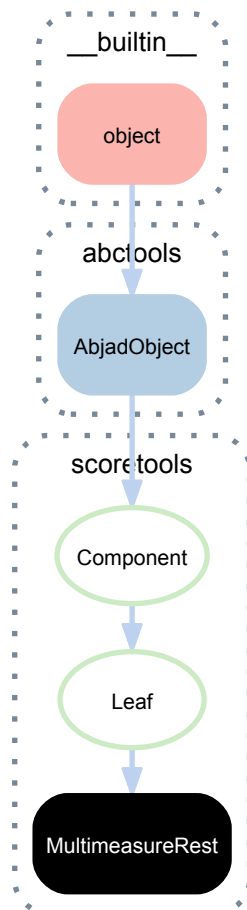
Measure.__str__()

String representation of measure.

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
>>> str(measure)
"|4/8 c'8 d'8 e'8 f'8|"
```

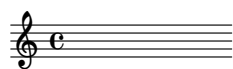
Returns string.

18.2.10 scoretools.MultimeasureRest



class scoretools.MultimeasureRest(*args)
A multimeasure rest.

```
>>> rest = scoretools.MultimeasureRest((1, 4))
>>> show(rest)
```



Multimeasure rests are immutable.

Bases

- `scoretools.Leaf`
- `scoretools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

Read/write properties

(`Leaf`) **.written_duration**

Written duration of leaf.

Set to duration.

Returns duration.

Special methods

(`Component`) **.__copy__** (*args)

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

Returns new component.

(`AbjadObject`) **.__eq__** (expr)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(`Component`) **.__format__** (format_specification='')

Formats component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

(`Component`) **.__illustrate__** ()

Illustrates component.

Returns LilyPond file.

(`Component`) **.__mul__** (n)

Copies component *n* times and detaches spanners.

Returns list of new components.

(`AbjadObject`) **.__ne__** (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(`Component`) **.__repr__** ()

Gets interpreter representation of leaf.

Returns string.

(`Component`) **.__rmul__** (n)

Copies component *n* times and detach spanners.

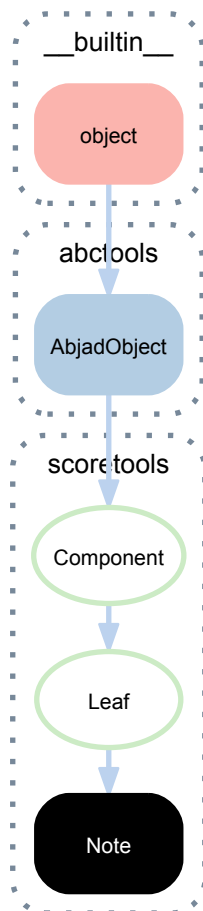
Returns list of new components.

(`Leaf`) **.__str__** ()

String representation of leaf.

Returns string.

18.2.11 scoretools.Note



class `scoretools.Note` (*args)
A note.

```
>>> note = Note("cs' 8.")
>>> measure = Measure((3, 16), [note])
>>> show(measure)
```



Bases

- `scoretools.Leaf`
- `scoretools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

Read/write properties

`Note.note_head`

Gets and sets note head of note.

Gets note head:

```
>>> note = Note(13, (3, 16))
>>> note.note_head
NoteHead("cs'")
```

Sets note head:

```
>>> note = Note(13, (3, 16))
>>> note.note_head = 14
>>> note
Note("d''8.")
```

Returns note head.

Note.**written_duration**

Gets and sets written duration of note.

Gets written duration of note.

```
>>> note = Note("c'4")
>>> note.written_duration
Duration(1, 4)
```

Sets written duration of note:

```
>>> note.written_duration = Duration(1, 16)
>>> note.written_duration
Duration(1, 16)
```

Returns duration

Note.**written_pitch**

Gets and sets written pitch of note.

Gets written pitch of note.

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch
NamedPitch("cs'")
```

Sets written pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch = 14
>>> note
Note("d''8.")
```

Returns named pitch.

Special methods

(Component).**__copy__**(*args)

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

Returns new component.

(AbjadObject).**__eq__**(expr)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(format_specification='')

Formats component.

Set *format_specification* to '', 'lilypond' or 'storage'.

Returns string.

(Component) .**__illustrate__**()
 Illustrates component.
 Returns LilyPond file.

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

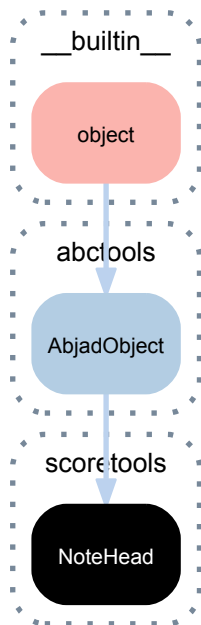
(AbjadObject) .**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(Component) .**__repr__**()
 Gets 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.

18.2.12 scoretools.NoteHead



```
class scoretools.NoteHead (written_pitch=None, client=None, is_cautionary=False,  

                           is_forced=False, tweak_pairs=())
```

A note head.

```
>>> note_head = scoretools.NoteHead(13)
>>> note_head
NoteHead("cs'')
```

Note heads are immutable.

Bases

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

Read-only properties

`NoteHead.client`

Client of note-head.

```
>>> note_head.client is None
True
```

Returns note, chord or none.

`NoteHead.named_pitch`

Named pitch of note-head.

```
>>> note_head = scoretools.NoteHead("cs' ")
>>> note_head.named_pitch
NamedPitch("cs' ")
```

Returns named pitch.

`NoteHead.tweak`

LilyPond tweak reservoir of note-head.

```
>>> note_head = scoretools.NoteHead("cs' ")
>>> note_head.tweak
LilyPondNameManager()
```

Returns LilyPond tweak reservoir.

Read/write properties

`NoteHead.is_cautionary`

Gets and sets cautionary accidental flag.

Gets cautionary accidental flag:

```
>>> note_head = scoretools.NoteHead("cs' ")
>>> note_head.is_cautionary
False
```

Sets cautionary accidental flag:

```
>>> note_head = scoretools.NoteHead("cs' ")
>>> note_head.is_cautionary = True
```

Returns boolean.

`NoteHead.is_forced`

Gets and sets forced accidental flag.

Gets forced accidental flag:

```
>>> note_head = scoretools.NoteHead("cs' ")
>>> note_head.is_forced
False
```

Sets forced accidental flag:

```
>>> note_head = scoretools.NoteHead("cs' ")
>>> note_head.is_forced = True
```

Returns boolean.

NoteHead.written_pitch

Gets and sets written pitch of note-head.

Gets written pitch of note-head:

```
>>> note_head = scoretools.NoteHead("cs' ")
>>> note_head.written_pitch
NamedPitch("cs' ")
```

Sets written pitch of note-head:

```
>>> note_head = scoretools.NoteHead("cs' ")
>>> note_head.written_pitch = "d' "
>>> note_head.written_pitch
NamedPitch("d' ")
```

Returns named pitch.

Special methods

NoteHead.__copy__(**args*)

Copies note-head.

```
>>> import copy
>>> copy.copy(note_head)
NoteHead("cs' ")
```

Returns new note-head.

NoteHead.__eq__(*expr*)

True when *expr* is a note-head with written pitch equal to that of this note-head. Otherwise false.

Returns boolean.

NoteHead.__format__(*format_specification*='')

Formats note-head.

Returns string.

NoteHead.__ge__(*other*)

$x._\text{ge}_(y) \iff x \geq y$

NoteHead.__gt__(*other*)

$x._\text{gt}_(y) \iff x > y$

NoteHead.__le__(*other*)

$x._\text{le}_(y) \iff x \leq y$

NoteHead.__lt__(*expr*)

True when *expr* is a note-head with written pitch greater than that of this note-head. Otherwise false.

Returns boolean.

(AbjadObject). **__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

NoteHead.__repr__()

Gets interpreter representation of note-head.

```
>>> note_head
NoteHead("cs' ")
```

Returns string.

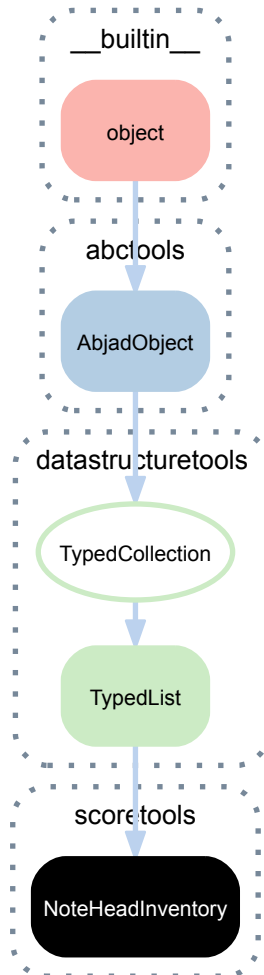
NoteHead.__str__()

String representation of note-head.

```
>>> str(note_head)
"cs' "
```

Returns string.

18.2.13 scoretools.NoteHeadInventory



class `scoretools.NoteHeadInventory` (*tokens=None, client=None*)
 An ordered list of note heads.

```
>>> chord = Chord([0, 1, 4], (1, 4))
>>> inventory = scoretools.NoteHeadInventory(
...     client=chord,
...     tokens=[11, 10, 9],
... )
```

```
>>> print format(inventory)
scoretools.NoteHeadInventory(
[
    scoretools.NoteHead(
        written_pitch=pitchtools.NamedPitch("a'"),
        is_cautionary=False,
        is_forced=False,
    ),
    scoretools.NoteHead(
        written_pitch=pitchtools.NamedPitch("bf'"),
        is_cautionary=False,
        is_forced=False,
    ),
    scoretools.NoteHead(
```

```

        written_pitch=pitchtools.NamedPitch("b'"),
        is_cautionary=False,
        is_forced=False,
    ),
]
)

```

Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`NoteHeadInventory.client`

The note head inventory's chord client.

`(TypedCollection).item_class`

Item class to coerce tokens into.

Read/write properties

`(TypedCollection).custom_identifier`

Gets and sets custom identifier of typed collection.

Returns string or none.

`(TypedList).keep_sorted`

Sorts collection on mutation if true.

Methods

`(TypedList).append(token)`

Changes *token* to item and appends.

```

>>> 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)`

Changes *token* to item and returns 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*)

Changes *tokens* to items and extends.

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

Returns none.

`NoteHeadInventory` **.get** (*pitch*)

Gets note head in note head inventory by *pitch*.

Example 1. Gets note head by pitch name:

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



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



Example 2. Gets note head by pitch number:

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



```
>>> note_head = chord.note_heads.get(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.

(`TypedList`) **.index** (*token*)

Changes *token* to item and returns index.

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

Returns index.

(`TypedList`) **.insert** (*i*, *token*)

Changes *token* to item and inserts.

```
>>> 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.

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

Aliases list.pop().

(TypedList) **.remove** (*token*)

Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) **.__format__** (*format_specification=''*)

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList) **.__getitem__** (*i*)

Aliases list.__getitem__().

(TypedList) **.__iadd__** (*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList(
[
    indicatortools.Dynamic(
        'ppp'
    ),
    indicatortools.Dynamic(
        'p'
    ),
    indicatortools.Dynamic(
        'mp'
    ),
    indicatortools.Dynamic(
        'mf'
    ),
    indicatortools.Dynamic(
        'fff'
    ),
],
item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList).**__reversed__**()

Aliases list.**__reversed__**().

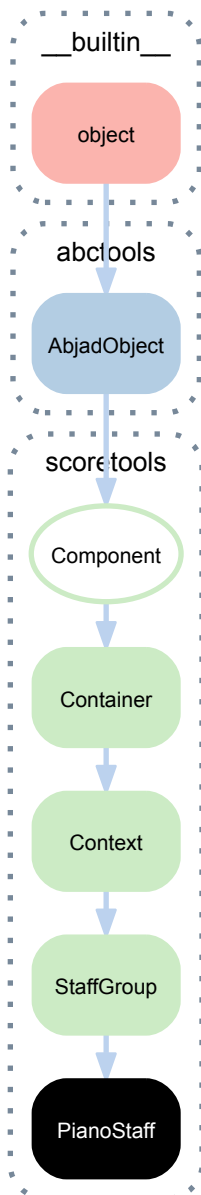
(TypedList).**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```

18.2.14 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`
- `scoretools.Context`
- `scoretools.Container`
- `scoretools.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')
>>> print format(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')
>>> print format(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

(Context) **.is_semantic**

True when context is semantic. Otherwise false.

Returns boolean.

Read/write properties

(Context) **.context_name**

Gets and sets context name of context.

Returns string.

(Context) **.is_nonsemantic**

Gets and sets nonsemantic voice flag.


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

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

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

Gets nonsemantic voice 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**

Gets and sets name of context.

Returns 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)
```



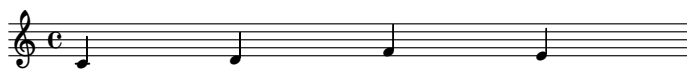
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
```

```
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

(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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component) .**__format__** (*format_specification*='')

Formats component.

Set *format_specification* to '', 'lilypond' or 'storage'.

Returns string.

(Container) .**__getitem__** (*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component) .**__illustrate__** ()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Context) .**__repr__** ()

Gets interpreter representation of context.

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

```

Returns string.

(Component) .**__rmul__** (*n*)

Copies component *n* times and detach spanners.

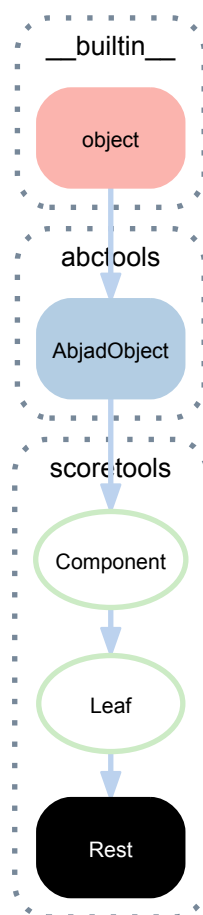
Returns list of new components.

(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.

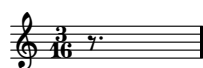
Returns none.

18.2.15 scoretools.Rest



class `scoretools.Rest` (*written_duration=None*)
 A rest.

```
>>> rest = Rest('r8.')
>>> measure = Measure((3, 16), [rest])
>>> show(measure)
```



Bases

- `scoretools.Leaf`
- `scoretools.Component`
- `abctools.AbjadObject`
- `__builtin__.object`

Read/write properties

(`Leaf`).**written_duration**

Written duration of leaf.

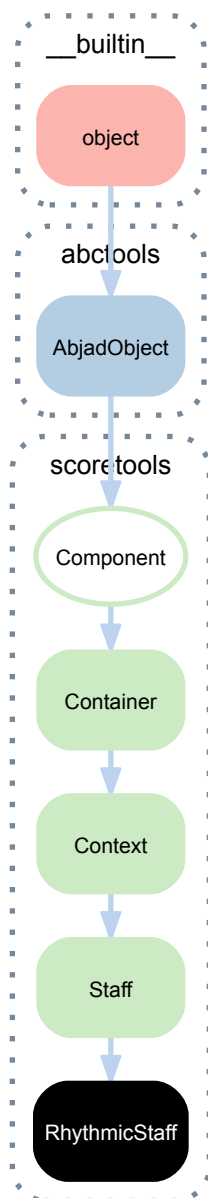
Set to duration.

Returns duration.

Special methods

- (Component) .**__copy__**(**args*)
 Copies component with indicators but without children of component or spanners attached to component.
 Returns new component.
- (AbjadObject) .**__eq__**(*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.
- (Component) .**__format__**(*format_specification*='')
 Formats component.
 Set *format_specification* to '', 'lilypond' or 'storage'.
 Returns string.
- (Component) .**__illustrate__**()
 Illustrates component.
 Returns LilyPond file.
- (Component) .**__mul__**(*n*)
 Copies component *n* times and detaches spanners.
 Returns list of new components.
- (AbjadObject) .**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.
- (Component) .**__repr__**()
 Gets 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.

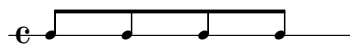
18.2.16 scoretools.RhythmicStaff



class `scoretools.RhythmicStaff` (*music=None*, *context_name='RhythmicStaff'*, *name=None*)
 A rhythmic staff.

```
>>> staff = scoretools.RhythmicStaff("c'8 d'8 e'8 f'8")
```

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



Returns `RhythmicStaff` instance.

Bases

- `scoretools.Staff`
- `scoretools.Context`
- `scoretools.Container`

- `scoretools.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')
>>> print format(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')
>>> print format(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

`(Context).is_semantic`

True when context is semantic. Otherwise false.

Returns boolean.

Read/write properties

`(Context).context_name`

Gets and sets context name of context.

Returns string.

`(Context).is_nonsemantic`

Gets and sets nonsemantic voice flag.

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

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

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

Gets nonsemantic voice 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**

Gets and sets name of context.

Returns 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)
```



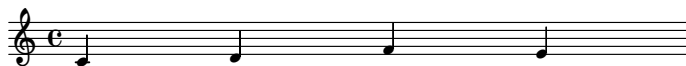
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

(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 indicators 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)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(format_specification='')

Formats component.

Set *format_specification* to '', 'lilypond' or 'storage'.

Returns string.

(Container).**__getitem__**(i)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component) .**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Context) .**__repr__**()

Gets interpreter representation of context.

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

```

Returns string.

(Component) .**__rmul__**(*n*)

Copies component *n* times and detach spanners.

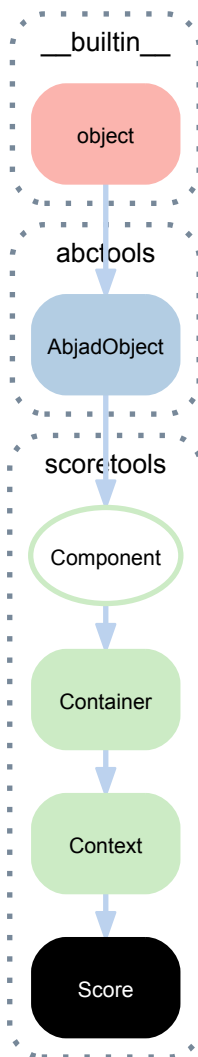
Returns list of new components.

(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.

18.2.17 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

- `scoretools.Context`
- `scoretools.Container`
- `scoretools.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')
>>> print format(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')
>>> print format(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

(Context).**is_semantic**

True when context is semantic. Otherwise false.

Returns boolean.

Read/write properties

(Context).**context_name**

Gets and sets context name of context.

Returns string.

(Context).**is_nonsemantic**

Gets and sets nonsemantic voice flag.

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

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

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

Gets nonsemantic voice 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**

Gets and sets name of context.

Returns string or none.

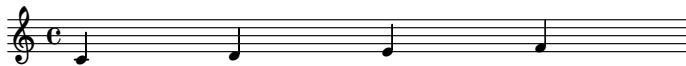
Methods

Score.**add_double_bar()**

Add double bar to end of score.

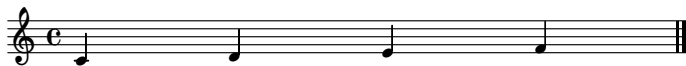
```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> score = Score([staff])
```

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



```
>>> score.add_double_bar()
BarLine('|.|')
```

```
>>> 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 format(markup, 'storage')
markuptools.Markup(
  (
    markuptools.MarkupCommand(
      'italic',
      markuptools.MarkupCommand(
        'right-column',
        ['Bremen - Boston - LA.', 'Jul 2010 - May 2011.'],
      )
    ),
  ),
  direction=Down,
)
```

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



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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

`(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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification*='')

Formats component.

Set *format_specification* to '', 'lilypond' or 'storage'.

Returns string.

(Container).**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component).**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Context).**__repr__**()

Gets interpreter representation of context.

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

```

Returns string.

(Component).**__rmul__**(*n*)

Copies component *n* times and detach spanners.

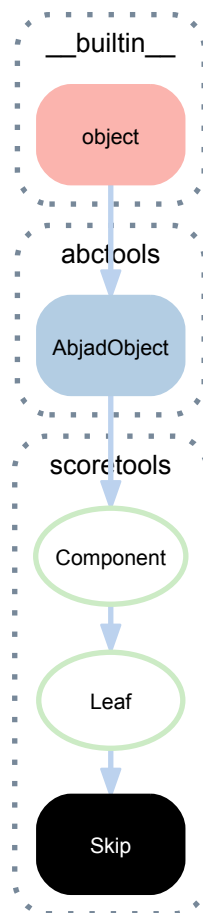
Returns list of new components.

(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.

18.2.18 scoretools.Skip



class scoretools.**Skip**(*arg*)
A LilyPond skip.

```
>>> skip = scoretools.Skip((3, 16))
>>> skip
Skip('s8.')
```

Bases

- scoretools.Leaf
- scoretools.Component

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

Read/write properties

(Leaf) **.written_duration**

Written duration of leaf.

Set to duration.

Returns duration.

Special methods

(Component) **.__copy__**(*args)

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

Returns new component.

(AbjadObject) **.__eq__**(expr)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component) **.__format__**(format_specification='')

Formats component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

(Component) **.__illustrate__**()

Illustrates component.

Returns LilyPond file.

(Component) **.__mul__**(n)

Copies component *n* times and detaches spanners.

Returns list of new components.

(AbjadObject) **.__ne__**(expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Component) **.__repr__**()

Gets interpreter representation of leaf.

Returns string.

(Component) **.__rmul__**(n)

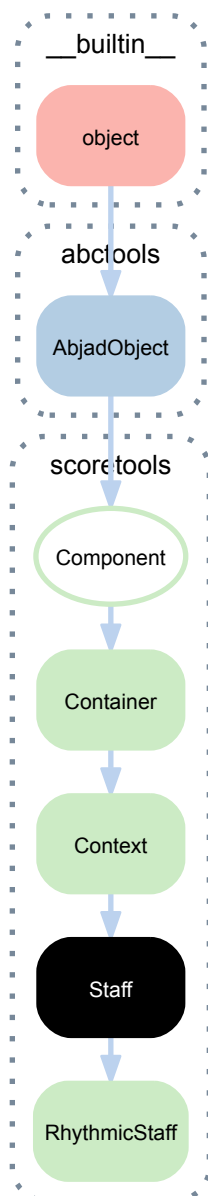
Copies component *n* times and detach spanners.

Returns list of new components.

(Leaf) **.__str__**()

String representation of leaf.

Returns string.

18.2.19 `scoretools.Staff`

class `scoretools.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

- `scoretools.Context`
- `scoretools.Container`
- `scoretools.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')
>>> print format(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')
>>> print format(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

`(Context).is_semantic`

True when context is semantic. Otherwise false.

Returns boolean.

Read/write properties

`(Context).context_name`

Gets and sets context name of context.

Returns string.

`(Context).is_nonsemantic`

Gets and sets nonsemantic voice flag.

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

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

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

Gets nonsemantic voice 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**

Gets and sets name of context.

Returns 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)
```



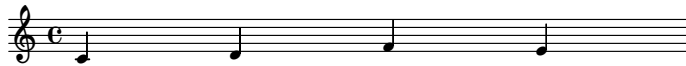
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

(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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification=''*)

Formats component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

(Container).**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component) .**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Context) .**__repr__**()

Gets interpreter representation of context.

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

```

Returns string.

(Component) .**__rmul__**(*n*)

Copies component *n* times and detach spanners.

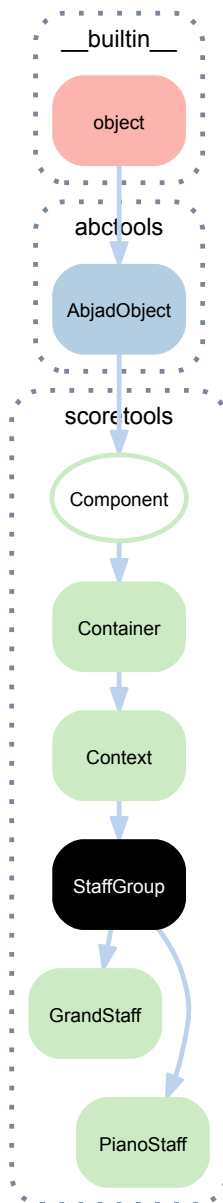
Returns list of new components.

(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.

18.2.20 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

- `scoretools.Context`
- `scoretools.Container`
- `scoretools.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')
>>> print format(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')
>>> print format(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

`(Context).is_semantic`

True when context is semantic. Otherwise false.

Returns boolean.

Read/write properties

`(Context).context_name`

Gets and sets context name of context.

Returns string.

`(Context).is_nonsemantic`

Gets and sets nonsemantic voice flag.

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

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

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

Gets nonsemantic voice 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**

Gets and sets name of context.

Returns 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)
```



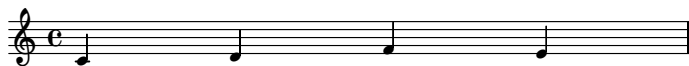
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, container[:])
>>> slur.direction = Down
>>> show(container)
```



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

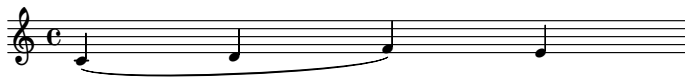


Returns none.

(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.

(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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification=''*)

Formats component.

Set *format_specification* to `'`, `'lilypond'` or `'storage'`.

Returns string.

(Container).**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component) .**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Context) .**__repr__**()

Gets interpreter representation of context.

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

```

Returns string.

(Component) .**__rmul__**(*n*)

Copies component *n* times and detach spanners.

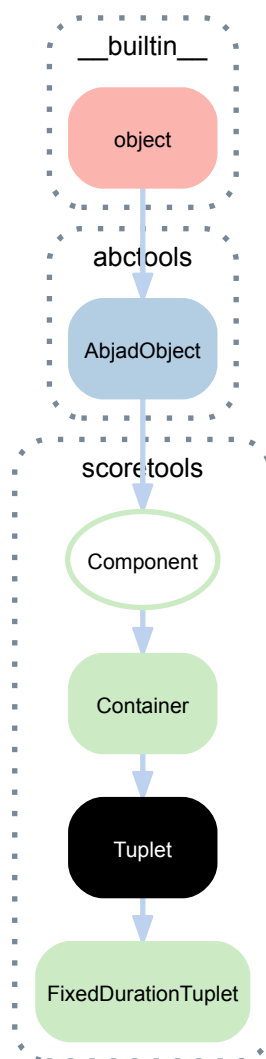
Returns list of new components.

(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.

18.2.21 scoretools.Tuplet



class `scoretools.Tuplet` (*multiplier=None, music=None*)
 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

- `scoretools.Container`
- `scoretools.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.multiplied_duration`

Multiplied duration of tuplet.

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.multiplied_duration
Duration(1, 4)
```

Returns duration.

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 1. Get tuplet invisibility status:

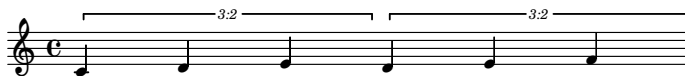
```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



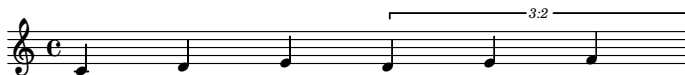
```
>>> tuplet.is_invisible is None
True
```

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)
```



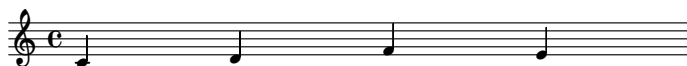
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, 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.

`(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*.

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.

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet
Tuplet(Multiplier(2, 3), "c'8 d'8 e'8")
```

```
>>> new_tuplet = tuplet.to_fixed_duration_tuplet()
>>> show(new_tuplet)
```



```
>>> new_tuplet
FixedDurationTuplet(Duration(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 diminished 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.

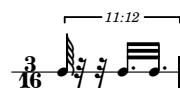
Make tupletted leaves strictly without dots when all *ratio* equal 1:

```
>>> 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 = scoretools.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 = scoretools.RhythmicStaff([measure])
>>> show(staff)
```



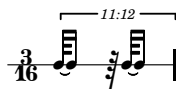
Interpret nonassignable *ratio* according to *decrease_durations_monotonically*:

```
>>> 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 = scoretools.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 = scoretools.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 = scoretools.RhythmicStaff([measure])
>>> show(staff)

```



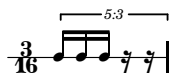
Example 3. Make diminished tuplet from *duration* and nonzero integer *ratio*.

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 = scoretools.RhythmicStaff([measure])
>>> show(staff)

```

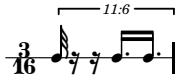


Allow tupletted leaves to return with dots when some *ratio* do not equal 1:

```

>>> 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 = scoretools.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 = scoretools.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 = scoretools.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 = scoretools.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(scoretools.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(scoretools.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(scoretools.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(scoretools.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(scoretools.RhythmicStaff([measure]))
```

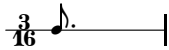


Example 1f. Change leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2, 3, 3, 4),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```

**Example 2a.** Change leaf to diminished tuplets with *ratio*:

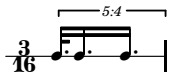
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```

**Example 2b.** Change leaf to diminished tuplets with *ratio*:

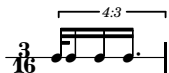
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```

**Example 2c.** Change leaf to diminished tuplets with *ratio*:

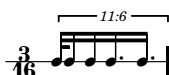
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> show(scoretools.RhythmicStaff([measure]))
```

**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(scoretools.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(scoretools.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(scoretools.RhythmicStaff([measure]))
```



Returns tuplet.

`Tuplet.from_nonreduced_ratio_and_nonreduced_fraction` (*ratio*, *fraction*)

Makes tuplet from nonreduced *ratio* and nonreduced *fraction*.

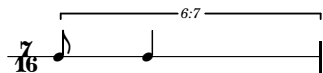
Example 1. Make container when no prolotion is necessary:

```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = scoretools.RhythmicStaff([measure])
>>> show(staff)
```

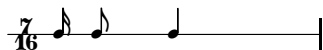


Example 2. Make fixed-duration tuplet when prolotion is necessary:

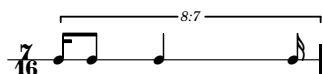
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = scoretools.RhythmicStaff([measure])
>>> show(staff)
```



```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = scoretools.RhythmicStaff([measure])
>>> show(staff)
```

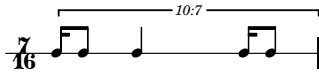


```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = scoretools.RhythmicStaff([measure])
>>> show(staff)
```

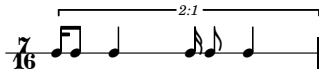


```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1, 2),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
```

```
>>> staff = scoretools.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 = scoretools.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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification*='')

Formats component.

Set *format_specification* to '', 'lilypond' or 'storage'.

Returns string.

(Container).**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component).**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

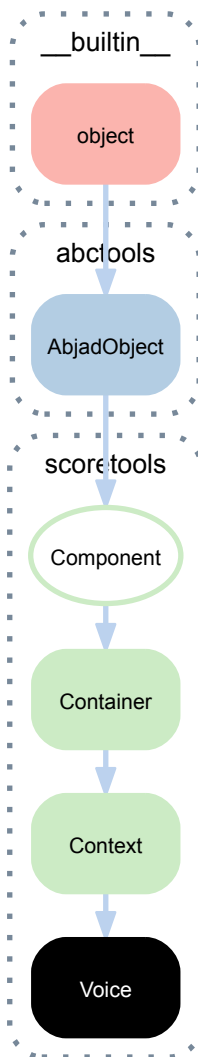
Tuplet .**__repr__**()
Gets interpreter representation of tuplet.
Returns string.

(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.

18.2.22 scoretools.Voice



class `scoretools.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

- `scoretools.Context`
- `scoretools.Container`
- `scoretools.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')
>>> print format(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')
>>> print format(staff)
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

`(Context).is_semantic`

True when context is semantic. Otherwise false.

Returns boolean.

Read/write properties

`(Context).context_name`

Gets and sets context name of context.

Returns string.

`(Context).is_nonsemantic`

Gets and sets nonsemantic voice flag.

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

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

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

Gets nonsemantic voice 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**

Gets and sets name of context.

Returns 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)
```



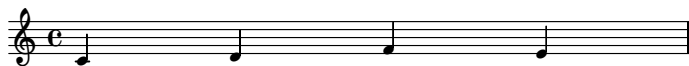
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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.Slur()
>>> attach(slur, 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.Slur()
>>> attach(slur, container[:])
>>> slur.direction = Down
>>> show(container)
```



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

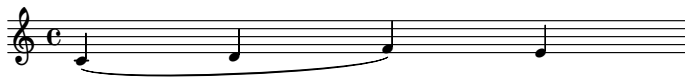


Returns none.

(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.

(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 indicators 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(Component).**__format__**(*format_specification=''*)

Formats component.

Set *format_specification* to *'*, *'lilypond'* or *'storage'*.

Returns string.

(Container).**__getitem__**(*i*)

Gets container *i*.

Traverses top-level items only.

Returns component.

(Component) .**__illustrate__**()

Illustrates component.

Returns LilyPond file.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(Context) .**__repr__**()

Gets interpreter representation of context.

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

```

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.

18.3 Functions

18.3.1 scoretools.append_spacer_skip_to_underfull_measure

`scoretools.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")
>>> detach(TimeSignature, measure)
>>> (TimeSignature((4, 12)),)
>>> new_time_signature = TimeSignature((5, 12))
>>> attach(new_time_signature, measure)
>>> measure.is_underfull
True

```

```
>>> scoretools.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*.

18.3.2 scoretools.append_spacer_skips_to_underfull_measures_in_expr

`scoretools.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)
>>> detach(TimeSignature, staff[1])
(TimeSignature((3, 8)),)
>>> new_time_signature = TimeSignature((4, 8))
>>> attach(new_time_signature, staff[1])
>>> detach(TimeSignature, staff[2])
(TimeSignature((3, 8)),)
>>> new_time_signature = TimeSignature((5, 8))
>>> attach(new_time_signature, staff[2])
>>> staff[1].is_underfull
True
>>> staff[2].is_underfull
True
```

```
>>> scoretools.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.

18.3.3 scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr

`scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr` (*expr*,
supple-
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)
```



```
>>> scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr(staff)
>>> show(staff)
```



Returns none.

18.3.4 scoretools.extend_measures_in_expr_and_apply_full_measure_tuplets

`scoretools.extend_measures_in_expr_and_apply_full_measure_tuplets` (*expr*,
supple-
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))]
>>> scoretools.extend_measures_in_expr_and_apply_full_measure_tuplets(
... staff, supplement)
```

Returns none.

18.3.5 scoretools.fill_measures_in_expr_with_full_measure_spacer_skips

`scoretools.fill_measures_in_expr_with_full_measure_spacer_skips` (*expr*, *iter-*
trl=None)

Fill measures in *expr* with full-measure spacer skips.

18.3.6 scoretools.fill_measures_in_expr_with_minimal_number_of_notes

`scoretools.fill_measures_in_expr_with_minimal_number_of_notes` (*expr*, *decrease_durations_monotonically=True*, *iterc-trl=None*)

Fill measures in *expr* with minimal number of notes that decrease durations monotonically:

```
>>> measure = Measure((5, 18), [])
```

```
>>> scoretools.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), [])
```

```
>>> scoretools.fill_measures_in_expr_with_minimal_number_of_notes(
...     measure, decrease_durations_monotonically=False)
```

Returns none.

18.3.7 scoretools.fill_measures_in_expr_with_repeated_notes

`scoretools.fill_measures_in_expr_with_repeated_notes` (*expr*, *written_duration*, *iterctrl=None*)

Fill measures in *expr* with repeated notes.

18.3.8 scoretools.fill_measures_in_expr_with_time_signature_denominator_notes

`scoretools.fill_measures_in_expr_with_time_signature_denominator_notes` (*expr*, *iterc-trl=None*)

Fill measures in *expr* with time signature denominator notes:

```
>>> staff = Staff([Measure((3, 4), []), Measure((3, 16), []), Measure((3, 8), [])])
>>> scoretools.fill_measures_in_expr_with_time_signature_denominator_notes(staff)
```

Delete existing contents of measures in *expr*.

Returns none.

18.3.9 scoretools.get_measure_that_starts_with_container

`scoretools.get_measure_that_starts_with_container` (*container*)

Get measure that starts with *container*.

Returns measure or none.

18.3.10 scoretools.get_measure_that_stops_with_container

`scoretools.get_measure_that_stops_with_container` (*container*)

Get measure that stops with *container*.

Returns measure or none.

18.3.11 `scoretools.get_next_measure_from_component`

`scoretools.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 |")
>>> scoretools.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 |")
>>> scoretools.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 |")
>>> scoretools.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 |")
>>> scoretools.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.

18.3.12 `scoretools.get_one_indexed_measure_number_in_expr`

`scoretools.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)
```



```
>>> scoretools.get_one_indexed_measure_number_in_expr(staff, 3)
Measure((2, 8), "g'8 a'8")
```

Note that measures number from 1.

18.3.13 `scoretools.get_previous_measure_from_component`

`scoretools.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 |")
>>> scoretools.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 |")
>>> scoretools.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 |")
>>> scoretools.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 |")
>>> scoretools.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.

18.3.14 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.

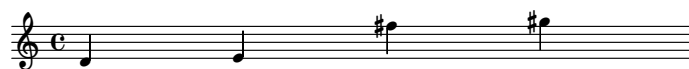
18.3.15 scoretools.make_leaves

`scoretools.make_leaves(pitches, durations, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None, metrical_hierarchy=None)`

Make leaves.

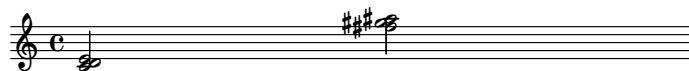
Example 1. Integer and string elements in *pitches* result in notes:

```
>>> pitches = [2, 4, 'F#5', 'G#5']
>>> duration = Duration(1, 4)
>>> leaves = scoretools.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 = scoretools.make_leaves(pitches, duration)
>>> staff = Staff(leaves)
>>> show(staff)
```



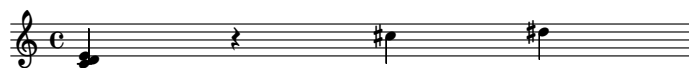
Example 3. None-valued elements in *pitches* result in rests:

```
>>> pitches = 4 * [None]
>>> durations = [Duration(1, 4)]
>>> leaves = scoretools.make_leaves(pitches, durations)
>>> staff = scoretools.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 = scoretools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```



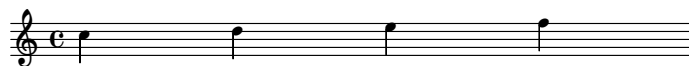
Example 5. Read *pitches* cyclically when the length of *pitches* is less than the length of *durations*:

```
>>> pitches = ['C5']
>>> durations = 2 * [Duration(3, 8), Duration(1, 8)]
>>> leaves = scoretools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```



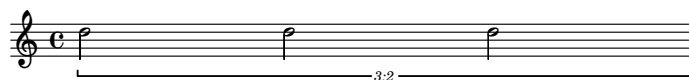
Example 6. Read *durations* cyclically when the length of *durations* is less than the length of *pitches*:

```
>>> pitches = "c' d' e' f'"
>>> durations = [Duration(1, 4)]
>>> leaves = scoretools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```



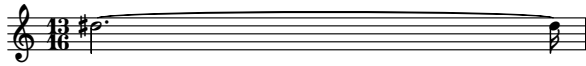
Example 7. Elements in *durations* with non-power-of-two denominators result in tuplet-nested leaves:

```
>>> pitches = ['D5']
>>> durations = [Duration(1, 3), Duration(1, 3), Duration(1, 3)]
>>> leaves = scoretools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```



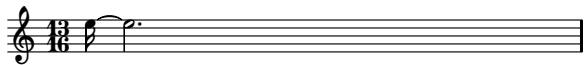
Example 8. Set *decrease_durations_monotonically* to true to return nonassignable durations tied from greatest to least:

```
>>> pitches = ['D#5']
>>> durations = [Duration(13, 16)]
>>> leaves = scoretools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> time_signature = TimeSignature((13, 16))
>>> attach(time_signature, 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 = scoretools.make_leaves(
...     pitches,
...     durations,
...     decrease_durations_monotonically=False,
... )
>>> staff = Staff(leaves)
>>> time_signature = TimeSignature((13, 16))
>>> attach(time_signature, 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 = scoretools.make_leaves(
...     pitches,
...     durations,
...     tie_rests=True,
... )
>>> staff = scoretools.RhythmicStaff(leaves)
>>> time_signature = TimeSignature((5, 8))
>>> attach(time_signature, staff)
>>> show(staff)
```



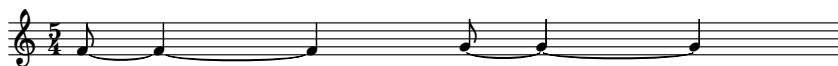
Example 11. Set *forbidden_written_duration* to avoid notes greater than or equal to a certain written duration:

```
>>> pitches = "f' g'"
>>> durations = [Duration(5, 8)]
>>> leaves = scoretools.make_leaves(
...     pitches,
...     durations,
...     forbidden_written_duration=Duration(1, 2),
... )
>>> staff = Staff(leaves)
>>> time_signature = TimeSignature((5, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



Example 12. You may set *forbidden_written_duration* and *decrease_durations_monotonically* together:

```
>>> pitches = "f' g'"
>>> durations = [Duration(5, 8)]
>>> leaves = scoretools.make_leaves(
...     pitches,
...     durations,
...     forbidden_written_duration=Duration(1, 2),
...     decrease_durations_monotonically=False,
... )
>>> staff = Staff(leaves)
>>> time_signature = TimeSignature((5, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



Returns selection of unincorporated leaves.

18.3.16 `scoretools.make_leaves_from_talea`

`scoretools.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 = scoretools.make_leaves_from_talea([3, -3, 5, -5], 16)
>>> staff = scoretools.RhythmicStaff(leaves)
>>> time_signature = TimeSignature((4, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



Example 2. Increase durations monotonically:

```
>>> leaves = scoretools.make_leaves_from_talea(
...     [3, -3, 5, -5], 16,
...     decrease_durations_monotonically=False)
>>> staff = scoretools.RhythmicStaff(leaves)
>>> time_signature = TimeSignature((4, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



Example 3. Forbid written durations greater than or equal to a half note:

```
>>> leaves = scoretools.make_leaves_from_talea(
...     [3, -3, 5, -5], 16,
...     forbidden_written_duration=Duration(1, 4))
>>> staff = scoretools.RhythmicStaff(leaves)
>>> time_signature = TimeSignature((4, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



Returns list of leaves.

18.3.17 `scoretools.make_multimeasure_rests`

`scoretools.make_multimeasure_rests` (*durations*)

Make multi-measure rests from *durations*:

```
>>> scoretools.make_multimeasure_rests([(4, 4), (7, 4)])
Selection(MultimeasureRest('R1'), MultimeasureRest('R1..'))
```

Returns list.

18.3.18 scoretools.make_multiplied_quarter_notes

`scoretools.make_multiplied_quarter_notes` (*pitches*, *multiplied_durations*)

Make quarter notes with *pitches* and *multiplied_durations*:

```
>>> args = [[0, 2, 4, 5], [(1, 4), (1, 5), (1, 6), (1, 7)]]
>>> scoretools.make_multiplied_quarter_notes(*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)]]
>>> scoretools.make_multiplied_quarter_notes(*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)]]
>>> scoretools.make_multiplied_quarter_notes(*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.

18.3.19 scoretools.make_notes

`scoretools.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*:

```
>>> scoretools.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*:

```
>>> scoretools.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:

```
>>> scoretools.make_notes([0], [(1, 16), (1, 12), (1, 8)])
Selection(Note("c'16"), Tuplet(Multiplier(2, 3), "c'8"), Note("c'8"))
```

Set *decrease_durations_monotonically*=True to express tied values in decreasing duration:

```
>>> scoretools.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:

```
>>> scoretools.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.

18.3.20 scoretools.make_notes_with_multiplied_durations

`scoretools.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)]]
>>> scoretools.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.

18.3.21 scoretools.make_percussion_note

`scoretools.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*.

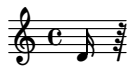
```
>>> leaves = scoretools.make_percussion_note(2, (1, 4), (1, 8))
>>> staff = Staff(leaves)
>>> show(staff)
```



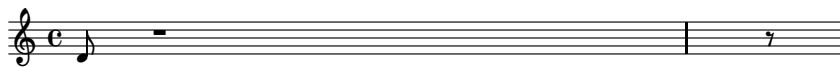
```
>>> leaves = scoretools.make_percussion_note(2, (1, 64), (1, 8))
>>> staff = Staff(leaves)
>>> show(staff)
```



```
>>> leaves = scoretools.make_percussion_note(2, (5, 64), (1, 8))
>>> staff = Staff(leaves)
>>> show(staff)
```



```
>>> leaves = scoretools.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.

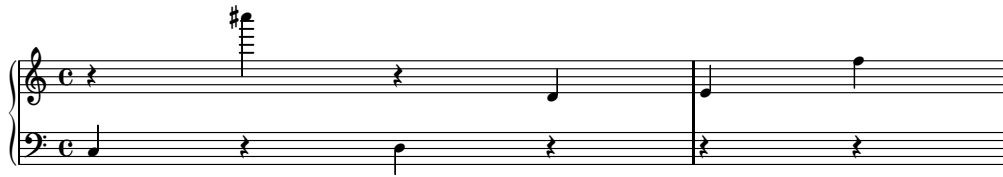
18.3.22 scoretools.make_piano_score_from_leaves

`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.

18.3.23 `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 = scoretools.make_notes(
...     [-12, -10, -8, -7, -5, 0, 2, 4, 5, 7],
...     [(1, 16)],
... )
>>> 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.

18.3.24 `scoretools.make_repeated_notes`

`scoretools.make_repeated_notes` (*count*, *duration=Duration(1, 8)*)

Make *count* repeated notes with note head-assignable *duration*:

```
>>> scoretools.make_repeated_notes(4)
Selection(Note("c'8"), Note("c'8"), Note("c'8"), Note("c'8"))
```

Make *count* repeated logical ties with tied *duration*:

```
>>> notes = scoretools.make_repeated_notes(2, (5, 16))
>>> voice = Voice(notes)
```

Make ad hoc tuplet holding *count* repeated notes with non-power-of-two *duration*:

```
>>> scoretools.make_repeated_notes(3, (1, 12))
Selection(Tuplet(Multiplier(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.

18.3.25 `scoretools.make_repeated_notes_from_time_signature`

`scoretools.make_repeated_notes_from_time_signature` (*time_signature*, *pitch*="c")

Make repeated notes from *time_signature*:

```
>>> scoretools.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*:

```
>>> scoretools.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.

18.3.26 `scoretools.make_repeated_notes_from_time_signatures`

`scoretools.make_repeated_notes_from_time_signatures` (*time_signatures*,
pitch="c")

Make repeated notes from *time_signatures*:

```
scoretools.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*:

```
>>> scoretools.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.

18.3.27 `scoretools.make_repeated_notes_with_shorter_notes_at_end`

`scoretools.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 = scoretools.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 = scoretools.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 = scoretools.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.

18.3.28 `scoretools.make_repeated_rests_from_time_signatures`

`scoretools.make_repeated_rests_from_time_signatures` (*time_signatures*)

Make repeated rests from *time_signatures*:

```
scoretools.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.

18.3.29 `scoretools.make_repeated_skips_from_time_signatures`

`scoretools.make_repeated_skips_from_time_signatures` (*time_signatures*)

Make repeated skips from *time_signatures*:

```
scoretools.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.

18.3.30 `scoretools.make_rests`

`scoretools.make_rests` (*durations*, *decrease_durations_monotonically=True*, *tie_parts=False*)

Make rests.

Make rests and decrease durations monotonically:

```
>>> scoretools.make_rests(
...     [(5, 16), (9, 16)],
...     decrease_durations_monotonically=True,
...     )
Selection(Rest('r4'), Rest('r16'), Rest('r2'), Rest('r16'))
```

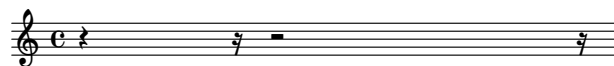
Makes rests and increase durations monotonically:

```
>>> scoretools.make_rests(
...     [(5, 16), (9, 16)],
...     decrease_durations_monotonically=False,
...     )
Selection(Rest('r16'), Rest('r4'), Rest('r16'), Rest('r2'))
```

Make tied rests:

```
>>> voice = Voice(scoretools.make_rests(
...     [(5, 16), (9, 16)],
...     tie_parts=True,
...     ))
```

```
>>> show(voice)
```



Returns list of rests.

18.3.31 `scoretools.make_rhythmic_sketch_staff`

`scoretools.make_rhythmic_sketch_staff` (*music*)

Make rhythmic staff with transparent *time_signature* and transparent bar lines.

18.3.32 scoretools.make_skips_with_multiplied_durations

`scoretools.make_skips_with_multiplied_durations` (*written_duration*, *multiplied_durations*)

Make *written_duration* skips with *multiplied_durations*:

```
>>> scoretools.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.

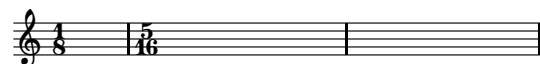
Returns list of skips.

18.3.33 scoretools.make_spacer_skip_measures

`scoretools.make_spacer_skip_measures` (*time_signatures*)

Make measures with full-measure spacer skips from *time_signatures*.

```
>>> measures = scoretools.make_spacer_skip_measures(
...     [(1, 8), (5, 16), (5, 16)])
>>> staff = Staff(measures)
>>> show(staff)
```



Returns selection of unincorporated measures.

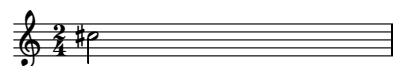
18.3.34 scoretools.make_tied_leaf

`scoretools.make_tied_leaf` (*kind*, *duration*, *decrease_durations_monotonically=True*, *forbid_den_written_duration=None*, *pitches=None*, *tie_parts=True*)

Make tied *kind* with *duration*.

Example 1. Make note:

```
>>> leaves = scoretools.make_tied_leaf(
...     Note,
...     Duration(1, 2),
...     pitches='C#5',
... )
>>> staff = Staff(leaves)
>>> time_signature = TimeSignature((2, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



Example 2. Make note and forbid half notes:

```
>>> leaves = scoretools.make_tied_leaf(
...     Note,
...     Duration(1, 2),
...     pitches='C#5',
...     forbidden_written_duration=Duration(1, 2),
... )
>>> staff = Staff(leaves)
>>> time_signature = TimeSignature((2, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



Example 3. Make tied note with half notes forbidden and durations decreasing monotonically:

```
>>> leaves = scoretools.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 = TimeSignature((9, 8))
>>> attach(time_signature, staff)
>>> show(staff)
```



Example 4. Make tied note with half notes forbidden and durations increasing monotonically:

```
>>> leaves = scoretools.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 = TimeSignature((9, 8))
>>> attach(time_signature, staff)
>>> show(staff)
```



Returns selection of unincorporated leaves.

18.3.35 `scoretools.move_full_measure_tuplet_prolation_to_measure_time_signature`

`scoretools.move_full_measure_tuplet_prolation_to_measure_time_signature` (*expr*)

Move prolation of full-measure tuplet to time signature of measure.

Measures usually become non-power-of-two as as result:

```
>>> t = Measure((2, 8), [scoretools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")])
>>> scoretools.move_full_measure_tuplet_prolation_to_measure_time_signature(t)
```

Returns none.

18.3.36 `scoretools.move_measure_prolation_to_full_measure_tuplet`

`scoretools.move_measure_prolation_to_full_measure_tuplet` (*expr*)

Move measure prolation 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.

18.3.37 `scoretools.replace_contents_of_measures_in_expr`

`scoretools.replace_contents_of_measures_in_expr` (*expr*, *new_contents*)

Replaces contents of measures in *expr* with *new_contents*.

```
>>> staff = Staff(scoretools.make_spacer_skip_measures(
...     [(1, 8), (3, 16)]))
>>> show(staff)
```



```
>>> notes = [Note("c'16"), Note("d'16"), Note("e'16"), Note("f'16")]
>>> scoretools.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.

18.3.38 scoretools.scale_measure_denominator_and_adjust_measure_contents

`scoretools.scale_measure_denominator_and_adjust_measure_contents` (*measure*,
factor)

Scales power-of-two *measure* to non-power-of-two measure with new denominator *factor*:

```
>>> measure = Measure((2, 8), "c'8 d'8")
>>> beam = spannertools.Beam()
>>> attach(beam, measure.select_leaves())
>>> show(measure)
```



```
>>> scoretools.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* duration.

Derives new *measure* multiplier.

Scales *measure* contents.

Picks best new time signature.

18.3.39 scoretools.set_always_format_time_signature_of_measures_in_expr

`scoretools.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.

18.3.40 `scoretools.set_measure_denominator_and_adjust_numerator`

`scoretools.set_measure_denominator_and_adjust_numerator` (*measure*, *denominator*)

Set *measure* time signature *denominator* and multiply time signature numerator accordingly:

```
>>> measure = Measure((3, 8), "c'8 d'8 e'8")
>>> beam = spannertools.Beam()
>>> attach(beam, measure.select_leaves())
```

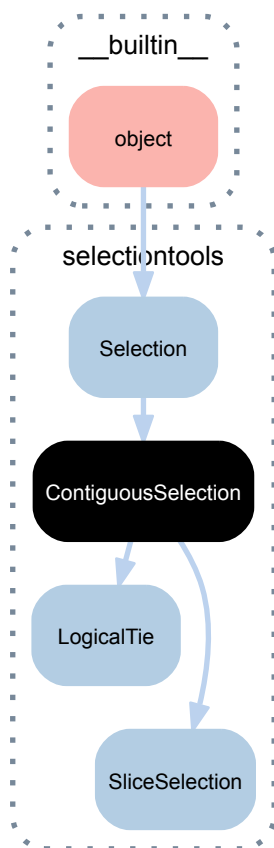
```
>>> scoretools.set_measure_denominator_and_adjust_numerator(measure, 16)
Measure((6, 16), "c'8 d'8 e'8")
```

Leave *measure* contents unchanged.

Return *measure*.

19.1 Concrete classes

19.1.1 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** (*prototype=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*)

Partitions components in selection by *durations* exactly.

Returns list of selections.

ContiguousSelection.**partition_by_durations_not_greater_than** (*durations, cyclic=False, in_seconds=False, overhang=False*)

Partitions components in selection by values of durations not greater than those in *durations*.

Returns list of selections.

ContiguousSelection.**partition_by_durations_not_less_than** (*durations, cyclic=False, in_seconds=False, overhang=False*)

Partitions components in selection by values of durations not less than those in *durations*.

Returns list of selections.

Special methods

ContiguousSelection.**__add__** (*expr*)

Adds *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).**__format__**(*format_specification*='')

Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(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*)

Adds selection to *expr*.

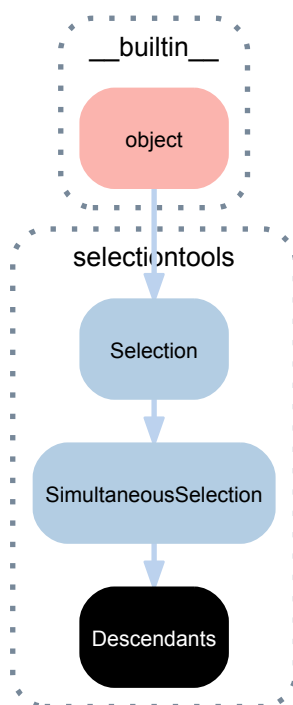
Returns new selection.

(Selection).**__repr__**()

Gets interpreter representation of selection.

Returns string.

19.1.2 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(prototype=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).__format__(format_specification='')`

Formats duration.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(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.

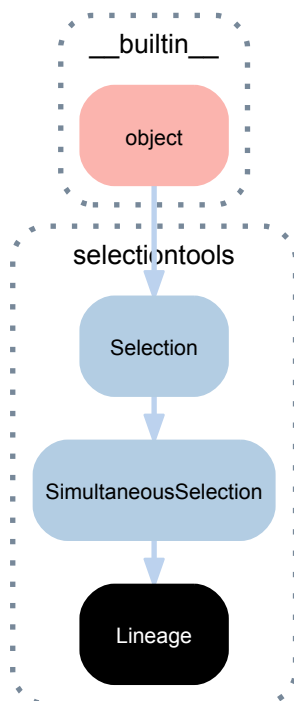
`(Selection).__radd__(expr)`

Concatenates selection to *expr*.

Returns newly created selection.

(Selection).**__repr__**()
 Gets interpreter representation of selection.
 Returns string.

19.1.3 selectiontools.Lineage



class selectiontools.**Lineage** (*component=None*)
 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(prototype=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).__format__(format_specification='')`

Formats duration.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(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.

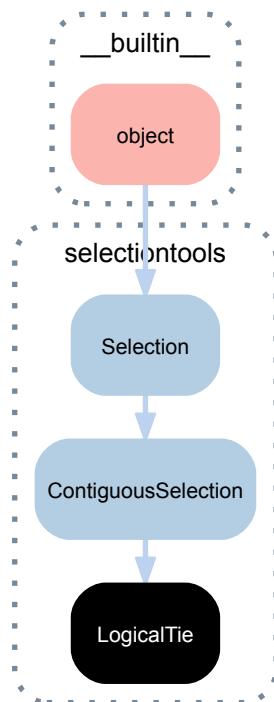
`(Selection).__radd__(expr)`

Concatenates selection to *expr*.

Returns newly created selection.

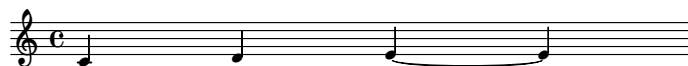
`(Selection).__repr__()`
 Gets interpreter representation of selection.
 Returns string.

19.1.4 selectiontools.LogicalTie



class selectiontools.**LogicalTie** (*music=None*)
 All the notes in a logical tie.

```
>>> staff = Staff("c' d' e' ~ e'")
>>> show(staff)
```



```
>>> inspect(staff[2]).get_logical_tie()
LogicalTie(Note("e'4"), Note("e'4"))
```

Bases

- selectiontools.ContiguousSelection
- selectiontools.Selection
- __builtin__.object

Read-only properties

LogicalTie.all_leaves_are_in_same_parent
 True when all leaves in logical tie are in same parent.
 Returns boolean.

LogicalTie.head
 Reference to element 0 in logical tie.

Returns component.

`LogicalTie.is_pitched`

True when logical tie head is a note or chord.

Returns boolean.

`LogicalTie.is_trivial`

True when length of logical tie is less than or equal to 1.

Returns boolean.

`LogicalTie.leaves`

Leaves in logical tie.

Returns tuple.

`LogicalTie.leaves_grouped_by_immediate_parents`

Leaves in logical tie grouped by immediate parents of leaves.

Returns list of lists.

`LogicalTie.tie_spanner`

Tie spanner governing logical tie.

Returns tie spanner.

`LogicalTie.written_duration`

Sum of written duration of all components in logical tie.

Returns duration.

Methods

`(Selection).get_duration(in_seconds=False)`

Gets duration of contiguous selection.

Returns duration.

`(Selection).get_spanners(prototype=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)
```

Partitions components in selection by *durations* exactly.

Returns list of selections.

```
(ContiguousSelection).partition_by_durations_not_greater_than(durations,
                                                             cyclic=False,
                                                             in_seconds=False,
                                                             overhang=False)
```

Partitions components in selection by values of durations not greater than those in *durations*.

Returns list of selections.

```
(ContiguousSelection).partition_by_durations_not_less_than(durations,
                                                           cyclic=False,
                                                           in_seconds=False,
                                                           overhang=False)
```

Partitions components in selection by values of durations not less than those in *durations*.

Returns list of selections.

LogicalTie.**to_tuplet** (*proportions*, *dotted=False*, *is_diminution=True*)

Change logical tie to tuplet.

Example 1. Change logical tie to diminished tuplet:

```
>>> staff = Staff(r"c'8 ~ c'16 cqs''4")
>>> crescendo = spannertools.Hairpin(descriptor='p < f')
>>> attach(crescendo, staff[:])
>>> override(staff).dynamic_line_spanner.staff_padding = 3
>>> time_signature = TimeSignature((7, 16))
>>> attach(time_signature, staff)
```

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



```
>>> logical_tie = inspect(staff[0]).get_logical_tie()
>>> logical_tie.to_tuplet([2, 1, 1, 1], is_diminution=True)
FixedDurationTuplet(Duration(3, 16), "c'8 c'16 c'16 c'16")
```

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



Example 2. Change logical tie to augmented tuplet:

```
>>> staff = Staff(r"c'8 ~ c'16 cqs''4")
>>> crescendo = spannertools.Hairpin(descriptor='p < f')
>>> attach(crescendo, staff[:])
>>> override(staff).dynamic_line_spanner.staff_padding = 3
>>> time_signature = TimeSignature((7, 16))
>>> attach(time_signature, staff)
```

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




```
>>> logical_tie = inspect(staff[0]).get_logical_tie()
>>> logical_tie.to_tuplet([2, 1, 1, 1], is_diminution=False)
FixedDurationTuplet(Duration(3, 16), "c' 16 c' 32 c' 32 c' 32")
```

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



Returns tuplet.

Special methods

(ContiguousSelection).**__add__**(*expr*)

Adds *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).**__format__**(*format_specification*='')

Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(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*)

Adds selection to *expr*.

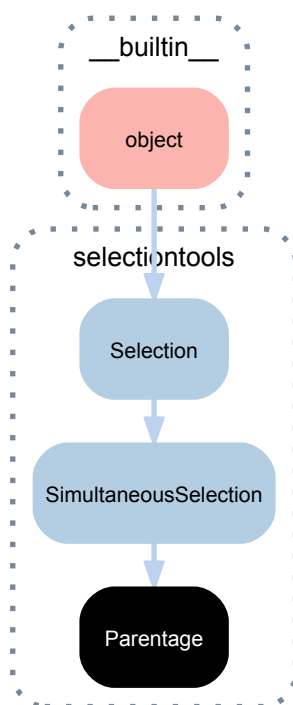
Returns new selection.

(Selection).**__repr__**()

Gets interpreter representation of selection.

Returns string.

19.1.5 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 = Clef('bass')
>>> attach(clef, 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

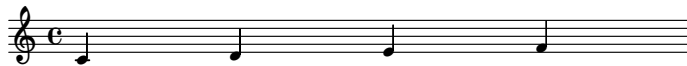
True when component has no parent. Otherwise false.

Returns boolean.

Parentage.logical_voice

Logical voice 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()
>>> logical_voice = parentage.logical_voice
```

```
>>> for key, value in logical_voice.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)
```



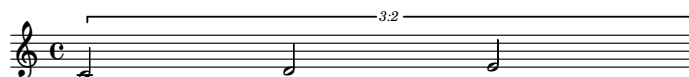
```
>>> leaves = score.select_leaves(allow_discontiguous_leaves=True)
>>> for leaf in leaves:
...     parentage = inspect(leaf).get_parentage()
...     leaf, parentage.score_index
...
(Note("c'2"), (0, 0, 0))
(Note("b'2"), (0, 0, 1))
(Note("a'2"), (0, 0, 2))
(Note("c'2"), (1, 0))
(Note("d'2"), (1, 1))
```

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**(*prototype=None*)
Gets first instance of *prototype* in parentage.

Returns component or none.

(Selection).**get_spanners**(*prototype=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).**__format__**(*format_specification*='')

Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(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*.

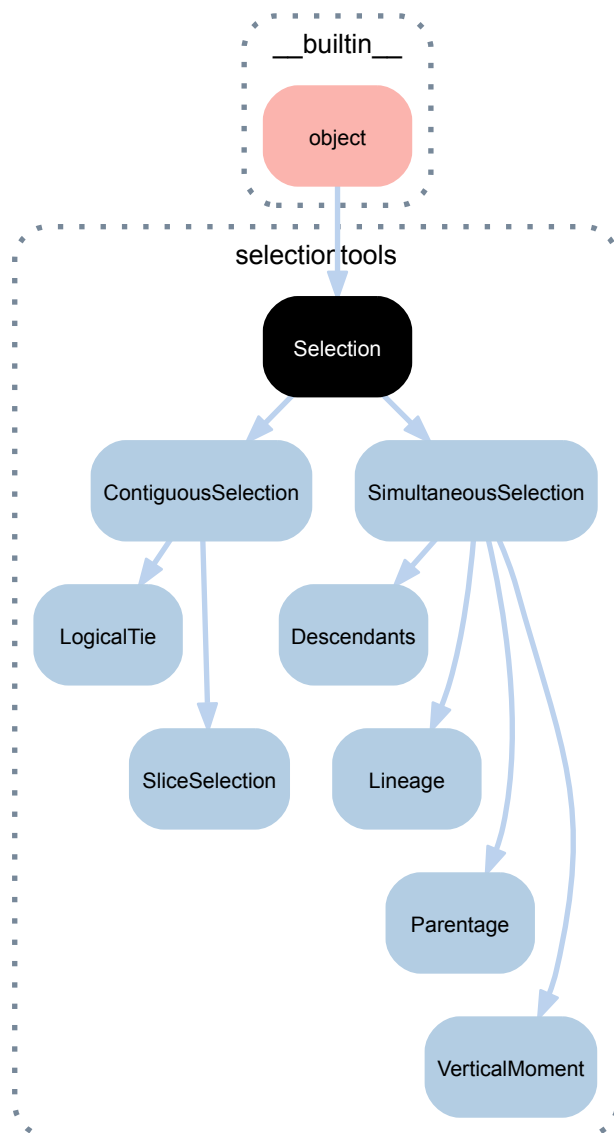
Returns newly created selection.

(Selection).**__repr__**()

Gets interpreter representation of selection.

Returns string.

19.1.6 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` (*prototype=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.__format__(format_specification='')`

Formats duration.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`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*.

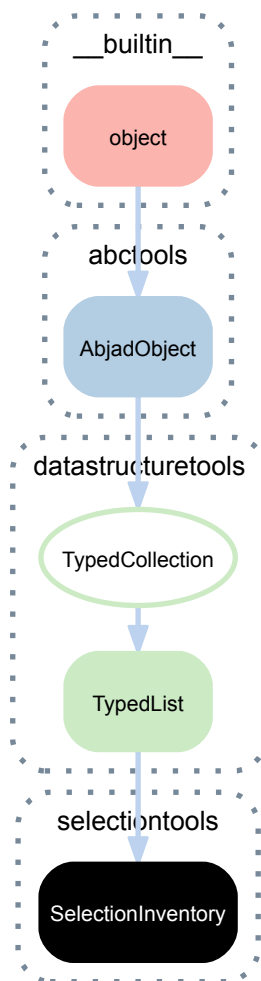
Returns newly created selection.

`Selection.__repr__()`

Gets interpreter representation of selection.

Returns string.

19.1.7 selectiontools.SelectionInventory



class selectiontools.**SelectionInventory** (*tokens=None*, *item_class=None*,
keep_sorted=None, *custom_identifier=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.

Read/write properties

(TypedCollection).**custom_identifier**
 Gets and sets custom identifier of typed collection.

Returns string or none.

(TypedList) **.keep_sorted**

Sorts collection on mutation if true.

Methods

(TypedList) **.append** (*token*)

Changes *token* to item and appends.

```
>>> 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*)

Changes *token* to item and returns 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*)

Changes *tokens* to items and extends.

```
>>> 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*)

Changes *token* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *token*)

Changes *token* to item and inserts.

```
>>> 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.

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

Aliases list.pop().

(TypedList) **.remove** (*token*)

Changes *token* to item and removes.

```
>>> 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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

(TypedCollection) **.__eq__** (*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) **.__format__** (*format_specification=''*)

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList) **.__getitem__** (*i*)

Aliases list.__getitem__().

(TypedList) **.__iadd__** (*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList (
  [
    indicatortools.Dynamic(
      'ppp'
    ),
    indicatortools.Dynamic(
      'p'
    ),
    indicatortools.Dynamic(
      'mp'
    ),
    indicatortools.Dynamic(
      'mf'
    ),
    indicatortools.Dynamic(
      'fff'
    ),
  ],
  item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList).**__reversed__**()

Aliases list.**__reversed__**().

(TypedList).**__setitem__**(*i, expr*)

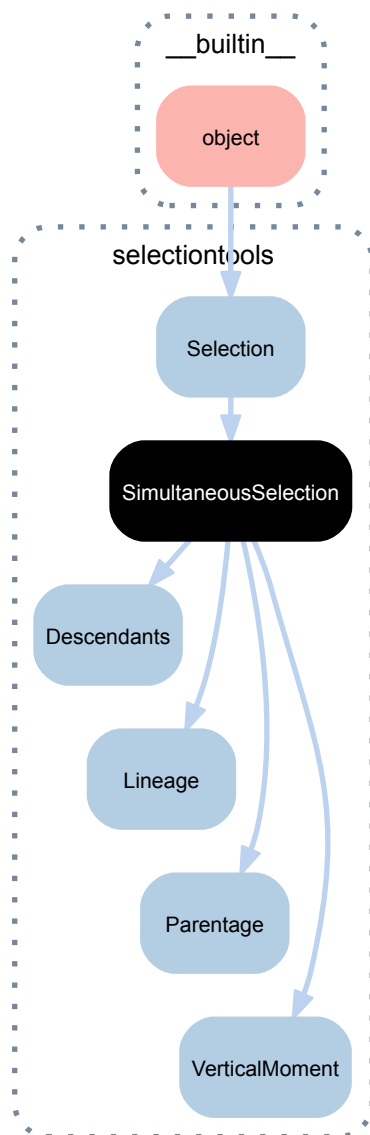
Changes tokens in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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,
)
```

19.1.8 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(prototype=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).__format__(format_specification='')`

Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(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*.

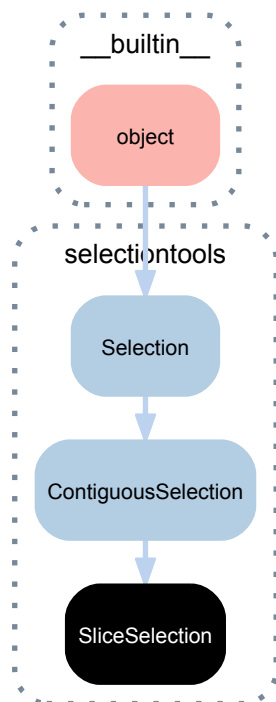
Returns newly created selection.

`(Selection).__repr__()`

Gets interpreter representation of selection.

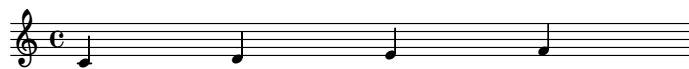
Returns string.

19.1.9 selectiontools.SliceSelection



class selectiontools.**SliceSelection** (*music=None*)
 A time-contiguous selection of components all in the same parent.

```
>>> 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** (*prototype=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*)

Partitions components in selection by *durations* exactly.

Returns list of selections.

(ContiguousSelection).**partition_by_durations_not_greater_than**(*durations*,
cyclic=False,
in_seconds=False,
overhang=False)

Partitions components in selection by values of durations not greater than those in *durations*.

Returns list of selections.

(ContiguousSelection).**partition_by_durations_not_less_than**(*durations*,
cyclic=False,
in_seconds=False,
overhang=False)

Partitions components in selection by values of durations not less than those in *durations*.

Returns list of selections.

Special methods

(ContiguousSelection).**__add__**(*expr*)

Adds *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).**__format__**(*format_specification=''*)

Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(Selection) .**__getitem__**(*expr*)
 Gets item *expr* from selection.
 Returns component from selection.

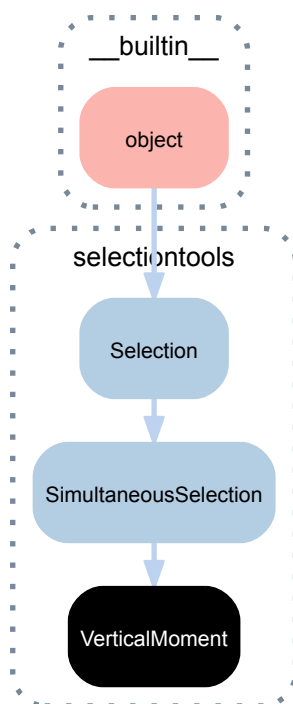
(Selection) .**__len__**()
 Number of components in selection.
 Returns nonnegative integer.

(Selection) .**__ne__**(*expr*)
 True when selection does not equal *expr*. Otherwise false.
 Returns boolean.

(ContiguousSelection) .**__radd__**(*expr*)
 Adds selection to *expr*.
 Returns new selection.

(Selection) .**__repr__**()
 Gets interpreter representation of selection.
 Returns string.

19.1.10 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 iterate(score).by_vertical_moment():
...     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 previous 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(prototype=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)`

True when *expr* is a vertical moment with the same components as this vertical moment. Otherwise false.

Returns boolean.

`(Selection).__format__(format_specification='')`

Formats duration.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Selection).__getitem__(expr)`

Gets item *expr* from selection.

Returns component from selection.

`VerticalMoment.__hash__()`

Hases vertical moment.

Returns integer.

`VerticalMoment.__len__()`

Length of vertical moment.

Defined equal to the number of components in vertical moment.

Returns nonnegative integer.

`VerticalMoment.__ne__(expr)`

True when *expr* does not equal this vertical moment. Otherwise false.

Returns boolean.

`(Selection).__radd__(expr)`

Concatenates selection to *expr*.

Returns newly created selection.

`VerticalMoment.__repr__()`

Gets interpreter representation of vertical moment.

Returns string.

20.1 Functions

20.1.1 `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
```

Otherwise false:

```
>>> sequencetools.all_are_assignable_integers('foo')
False
```

Returns boolean.

20.1.2 `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
```

Otherwise false:

```
>>> sequencetools.all_are_equal(17)
False
```

Returns boolean.

20.1.3 `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.

20.1.4 `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.

20.1.5 `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.

20.1.6 `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
```

Otherwise false:

```
>>> sequencetools.all_are_nonnegative_integer_powers_of_two(17)
False
```

Returns boolean.

20.1.7 `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.

20.1.8 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
```

Otherwise false:

```
>>> sequencetools.all_are_numbers(17)
False
```

Returns boolean.

20.1.9 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
```

Otherwise false:

```
>>> sequencetools.all_are_pairs('foo')
False
```

Returns boolean.

20.1.10 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
```

Otherwise false:

```
>>> sequencetools.all_are_pairs_of_types('foo', float, str)
False
```

Returns boolean.

20.1.11 `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.

20.1.12 `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.

20.1.13 `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
```

Otherwise false:

```
>>> sequencetools.all_are_unequal(17)
False
```

Returns boolean.

20.1.14 `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.

20.1.15 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.

20.1.16 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.

20.1.17 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.

20.1.18 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.

20.1.19 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.

20.1.20 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.

20.1.21 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.

20.1.22 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.

20.1.23 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.

20.1.24 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.

20.1.25 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.

20.1.26 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.

20.1.27 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.

20.1.28 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.

20.1.29 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.

20.1.30 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.

20.1.31 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.

20.1.32 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.

20.1.33 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.

20.1.34 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.

20.1.35 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.

20.1.36 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.

20.1.37 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.

20.1.38 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.

20.1.39 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.

20.1.40 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.

20.1.41 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 $l[i] \leq \max(l[:i]) + 1$ for $1 \leq i \leq \text{len}(l)$.

Returns boolean.

20.1.42 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.

20.1.43 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.

20.1.44 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.

20.1.45 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.

20.1.46 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.

20.1.47 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.

20.1.48 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.

20.1.49 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.

20.1.50 `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.

20.1.51 `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.

20.1.52 `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.

20.1.53 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.

20.1.54 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.

20.1.55 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.

20.1.56 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.

20.1.57 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.

20.1.58 `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.

20.1.59 `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.

20.1.60 `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.

20.1.61 `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.

20.1.62 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.

20.1.63 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.

20.1.64 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.

20.1.65 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*.

20.1.66 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.

20.1.67 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.

20.1.68 `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.

20.1.69 `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.

20.1.70 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.

20.1.71 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.

20.1.72 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.

20.1.73 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.

20.1.74 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.

20.1.75 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.

20.1.76 `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.

20.1.77 `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.

20.1.78 `sequencetools.remap_sequence_by_range_pairs`

`sequencetools.remap_sequence_by_range_pairs` (*sequence*, *range_pairs*)

Remaps *sequence* by *range_pairs*:

```
>>> sequence = [9, 14, 4, 1, 7, 5, 9, 6, 2, 10, 15, 20, 8, 4, 0, 7]  
>>> range_pairs = [  
...     ((2, 10), (3, 5)),  
...     ((10, 20), (6, 8)),  
... ]  
>>> sequencetools.remap_sequence_by_range_pairs(sequence, range_pairs)  
[4, 7, 5, 1, 5, 3, 4, 4, 3, 7, 8, 7, 3, 5, 0, 5]
```

Returns newly created *sequence* object.

20.1.79 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.

20.1.80 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.

20.1.81 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.

20.1.82 sequencetools.repeat_runs_in_sequence_to_count

`sequencetools.repeat_runs_in_sequence_to_count(sequence, tokens)`

Repeat subruns in *sequence* according to *tokens*. The *tokens* input parameter must be a list of zero or more (start, length, count) triples. For every (start, length, count) token in *tokens*, 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, 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()`.

20.1.83 `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.

20.1.84 `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.

20.1.85 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.

20.1.86 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.

20.1.87 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.

20.1.88 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.

20.1.89 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.

20.1.90 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.

20.1.91 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.

20.1.92 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.

20.1.93 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.

20.1.94 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.

20.1.95 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.

20.1.96 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.

20.1.97 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.

20.1.98 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.

20.1.99 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.

20.1.100 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.

20.1.101 `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.

20.1.102 `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.

20.1.103 `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.

20.1.104 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.

20.1.105 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.

20.1.106 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.

20.1.107 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.

20.1.108 sequencetools.yield_all_partitions_of_sequence

`sequencetools.yield_all_partitions_of_sequence(sequence)`

Yields all partitions of *sequence*.

```
>>> sequence = [0, 1, 2, 3]
>>> result = sequencetools.yield_all_partitions_of_sequence(sequence)
>>> for partition in result:
...     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.

20.1.109 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.

20.1.110 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.

20.1.111 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.

20.1.112 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.

20.1.113 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.

20.1.114 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.

20.1.115 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.

20.1.116 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.

20.1.117 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*.

20.1.118 sequencetools.zip_sequences_without_truncation

`sequencetools.zip_sequences_without_truncation(*sequences)`

Zip *sequences* nontruncating:

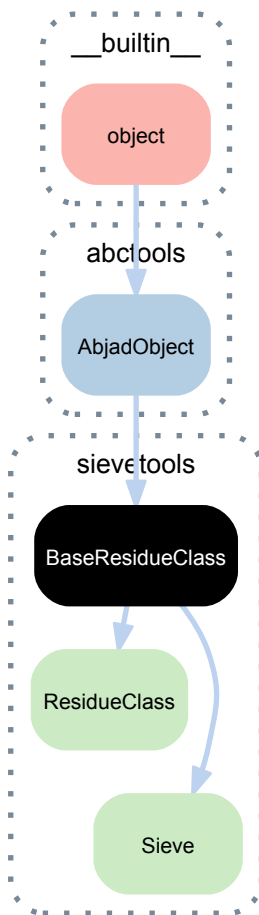
```
>>> sequencetools.zip_sequences_without_truncation(  
...     [1, 2, 3, 4], [11, 12, 13], [21, 22, 23])  
[(1, 11, 21), (2, 12, 22), (3, 13, 23), (4,)]
```

Lengths of the tuples returned may differ but will always be greater than or equal to 1.

Returns list of tuples.

21.1 Concrete classes

21.1.1 `sievetools.BaseResidueClass`



class `sievetools.BaseResidueClass`
Abstract base class for `ResidueClass` and `Sieve`.

Bases

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

Special methods

`BaseResidueClass.__and__(arg)`

Logical AND of residue class and *arg*.

Returns sieve.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`BaseResidueClass.__or__(arg)`

Logical OR of residue class and *arg*.

Returns sieve.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

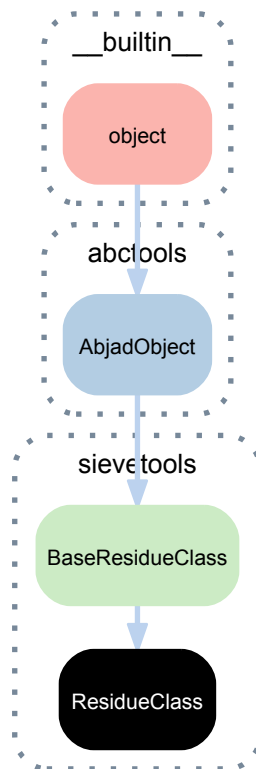
Returns string.

`BaseResidueClass.__xor__(arg)`

Logical XOR of residue class and *arg*.

Returns sieve.

21.1.2 sievetools.ResidueClass



class `sieve tools.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.

From the opening of Xenakis's *Psappha* for solo percussion:

```
>>> RC = sieve tools.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]
```

Bases

- `sieve tools.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.

```
>>> 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.

```
>>> 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)`

Logical AND of residue class and *arg*.

Returns sieve.

`ResidueClass.__eq__(expr)`

True when *expr* is a residue class with module and residue equal to those of this residue class. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`ResidueClass.__ge__(other)`

`x.__ge__(y) <==> x>=y`

`ResidueClass.__gt__(other)`

`x.__gt__(y) <==> x>y`

`ResidueClass.__le__(other)`
`x.__le__(y) <==> x<=y`

`ResidueClass.__lt__(expr)`
 True when *expr* is a residue class with module greater than that of this residue class. Also true when *expr* is a residue class with modulo equal to that of this residue class and with residue greater than that of this residue class. Otherwise false.

Returns boolean.

`ResidueClass.__ne__(expr)`
 True when *expr* is not equal to this residue class. Otherwise false.

Return boolean.

`(BaseResidueClass).__or__(arg)`
 Logical OR of residue class and *arg*.

Returns sieve.

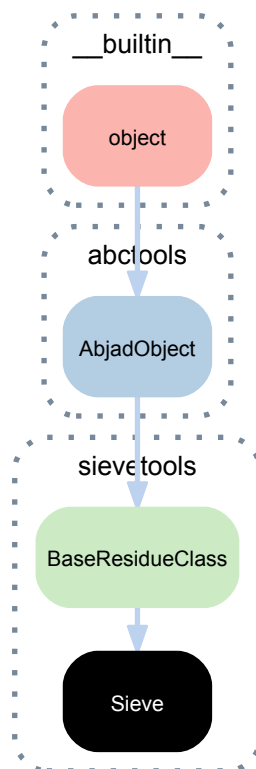
`(AbjadObject).__repr__()`
 Gets interpreter representation of Abjad object.

Returns string.

`(BaseResidueClass).__xor__(arg)`
 Logical XOR of residue class and *arg*.

Returns sieve.

21.1.3 sievetools.Sieve



class `sievetools.Sieve` (*rsc=None*, *logical_operator='or'*)
 A Xenakis sieve.

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.

```
>>> from abjad.tools.sievetools import ResidueClass
```

```
>>> sieve = ResidueClass(3, 0) | ResidueClass(2, 0)
>>> sieve.get_boolean_train(6)
[1, 0, 1, 1, 1, 0]
>>> sieve.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 it given, it is taken as the max range and min is assumed to be 0.

```
>>> sieve = ResidueClass(3, 0) | ResidueClass(2, 0)
>>> sieve.get_congruent_bases(6)
[0, 2, 3, 4, 6]
>>> sieve.get_congruent_bases(-6, 6)
[-6, -4, -3, -2, 0, 2, 3, 4, 6]
```

Returns list.

`Sieve.is_congruent_base(integer)`

True when *integer* is congruent to base in sieve. Otherwise false.

```
>>> sieve = ResidueClass(3, 0) | ResidueClass(2, 0)
>>> sieve.get_congruent_bases(6)
[0, 2, 3, 4, 6]
>>> sieve.is_congruent_base(12)
True
```

Otherwise false:

```
>>> sieve.is_congruent_base(13)
False
```

Returns boolean.

Static methods

`Sieve.from_cycle_tokens(*cycle_tokens)`

Makes 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]
```

```
>>> sieve = sievetools.Sieve.from_cycle_tokens(*cycle_tokens)
>>> print format(sieve)
sievetools.Sieve(
  rcs=[
    sievetools.ResidueClass(6, 0),
    sievetools.ResidueClass(6, 4),
    sievetools.ResidueClass(6, 5),
    sievetools.ResidueClass(10, 6),
    sievetools.ResidueClass(10, 7),
    sievetools.ResidueClass(10, 8),
  ],
  logical_operator='or',
)
```

Cycle token comprises *modulo*, *residues* and optional *offset*.

Special methods

`(BaseResidueClass).__and__(arg)`

Logical AND of residue class and *arg*.

Returns sieve.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(BaseResidueClass).__or__(arg)`

Logical OR of residue class and *arg*.

Returns sieve.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

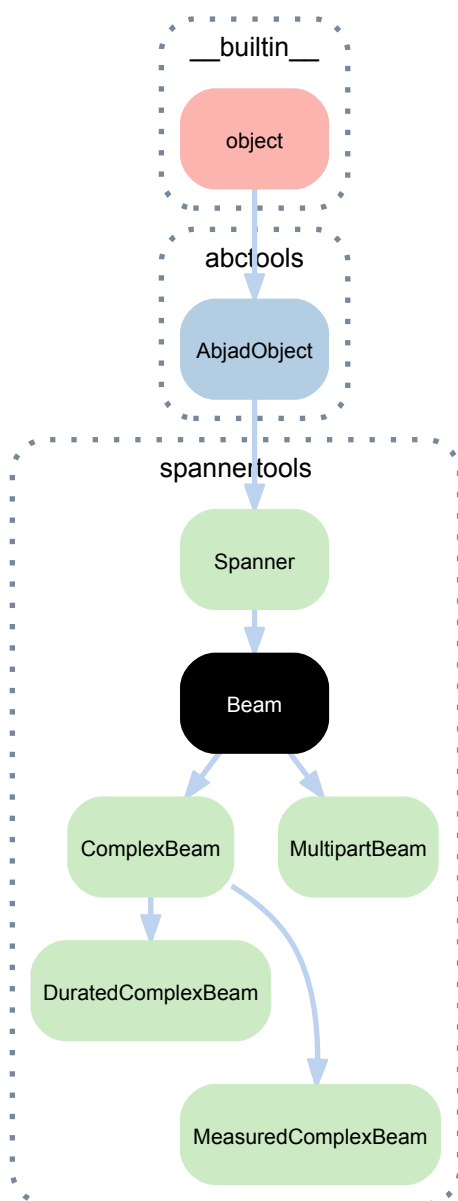
`(BaseResidueClass) .__xor__(arg)`

Logical XOR of residue class and *arg*.

Returns sieve.

22.1 Concrete classes

22.1.1 spannertools.Beam



class `spannertools.Beam` (*components=None, direction=None, overrides=None*)
A beam.

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'2")
>>> show(staff)
```



```
>>> beam = spannertools.Beam()
>>> attach(beam, staff[:4])
>>> show(staff)
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(Spanner) . **components**
Components in spanner.
Returns tuple.

(Spanner) . **leaves**
Leaves in spanner.
Returns tuple.

Read/write properties

Beam. **direction**
Gets and sets direction of beam.
Returns up or down.

Methods

(Spanner) . **append** (*component*)
Appends *component* to spanner.
Returns none.

(Spanner) . **append_left** (*component*)
Appends *component* to left of spanner.
Returns none.

(Spanner) . **detach** ()
Detaches spanner.
Returns none.

(Spanner) . **extend** (*components*)
Extends spanner with *components*.
Returns none.

(Spanner) **.extend_left** (*components*)
 Extends left of spanner with *components*.
 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.
 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*.
 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.
 Returns nonnegative integer.

(Spanner) **.pop** ()
 Pops rightmost component off of spanner.
 Returns component.

(Spanner) **.pop_left** ()
 Pops leftmost component off of spanner.
 Returns component.

Static methods

Beam.**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.Beam
...     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*) .__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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(*AbjadObject*) .__format__ (*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(*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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

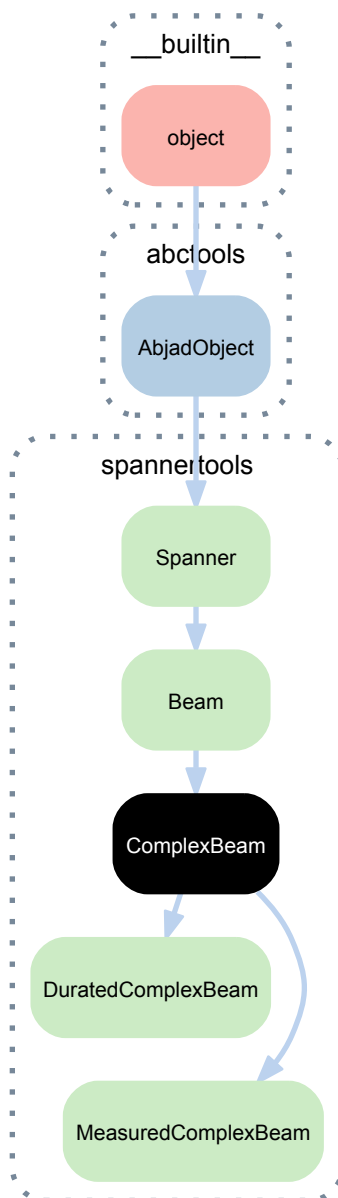
Returns boolean.

(*AbjadObject*) .__repr__ ()

Gets interpreter representation of Abjad object.

Returns string.

22.1.2 spannertools.ComplexBeam



class `spannertools.ComplexBeam` (*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)
```



```
>>> beam = spannertools.ComplexBeam()
>>> attach(beam, staff[:4])
>>> show(staff)
```



Bases

- `spannertools.Beam`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**
Components in spanner.

Returns tuple.

(Spanner) **.leaves**
Leaves in spanner.

Returns tuple.

Read/write properties

(Beam) **.direction**
Gets and sets direction of beam.

Returns up or down.

ComplexBeam **.lone**
Beam lone leaf and force beam nibs to left:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='left')
>>> attach(beam, note)
>>> show(note)
```



Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='right')
>>> attach(beam, note)
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='both')
>>> attach(beam, note)
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone=True)
>>> attach(beam, note)
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone=False)
>>> attach(beam, note)
```

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.
Returns none.
- (Spanner) . **append_left** (*component*)
Appends *component* to left of spanner.
Returns none.
- (Spanner) . **detach** ()
Detaches spanner.
Returns none.
- (Spanner) . **extend** (*components*)
Extends spanner with *components*.
Returns none.
- (Spanner) . **extend_left** (*components*)
Extends left of spanner with *components*.
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.
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*.
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.
Returns nonnegative integer.
- (Spanner) . **pop** ()
Pops rightmost component off of spanner.
Returns component.

(Spanner).**pop_left**()
 Pops leftmost component off of spanner.
 Returns component.

Static methods

(Beam).**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.Beam
...     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).**__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*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(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)`

Is true when Abjad object does not equal *expr*. Otherwise false.

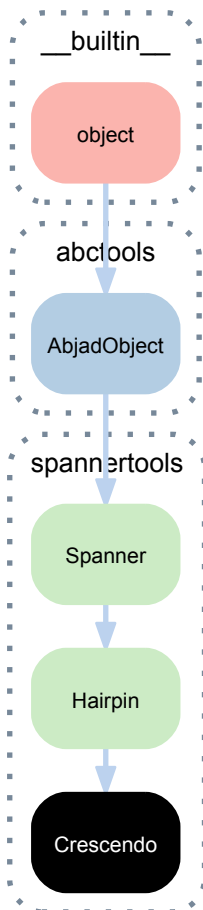
Returns boolean.

`(AbjadObject) .__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

22.1.3 spannertools.Crescendo



class `spannertools.Crescendo` (*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")
>>> show(staff)
```



```
>>> crescendo = spannertools.Crescendo(include_rests=True)
>>> attach(crescendo, staff[:])
>>> show(staff)
```



Abjad crescendo spanner that does not include rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> show(staff)
```



```
>>> crescendo = spannertools.Crescendo(include_rests=False)
>>> attach(crescendo, staff[:])
>>> show(staff)
```



Bases

- `spannertools.Hairpin`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**
Components in spanner.
Returns tuple.

(Spanner) **.leaves**
Leaves in spanner.
Returns tuple.

Read/write properties

(Hairpin) **.direction**
Gets and sets direction of hairpin.
Returns up or down.

(Hairpin) **.include_rests**
Gets and sets boolean setting to include rests in hairpin.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     include_rests=True,
... )
>>> attach(hairpin, staff[:])
>>> hairpin.include_rests
True
```

Sets include-rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     include_rests=True,
... )
>>> attach(hairpin, staff[:])
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Returns boolean.

(Hairpin).**shape_string**

Gets and sets hairpin shape string.

Gets hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.shape_string
'<'
```

Sets hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.shape_string = '>'
>>> hairpin.shape_string
'>'
```

Returns string.

(Hairpin).**start_dynamic_string**

Gets and sets start dynamic string of hairpin.

Gets start dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.start_dynamic_string
'p'
```

Sets start dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Returns string.

(Hairpin).**stop_dynamic_string**

Gets and sets stop dynamic string of hairpin.

Gets stop dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.stop_dynamic_string
'f'
```

Sets stop dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Returns string.

Methods

(Spanner) .**append**(*component*)

Appends *component* to spanner.

Returns none.

(Spanner) .**append_left**(*component*)

Appends *component* to left of spanner.

Returns none.

(Spanner) .**detach**()

Detaches spanner.

Returns none.

(Spanner) .**extend**(*components*)

Extends spanner with *components*.

Returns none.

(Spanner) .**extend_left**(*components*)

Extends left of spanner with *components*.

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.

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*.

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.

Returns nonnegative integer.

(Spanner) .**pop**()

Pops rightmost component off of spanner.

Returns component.

`(Spanner).pop_left()`
 Pops leftmost component off of spanner.
 Returns component.

Static methods

`(Hairpin).is_hairpin_shape_string(arg)`
 True when *arg* is a hairpin shape string. Otherwise false:

```
>>> spannertools.Hairpin.is_hairpin_shape_string('<')
True
```

Returns boolean.

`(Hairpin).is_hairpin_token(arg)`
 True when *arg* is a hairpin token. Otherwise false:

```
>>> spannertools.Hairpin.is_hairpin_token(('p', '<', 'f'))
True
```

```
>>> spannertools.Hairpin.is_hairpin_token(('f', '<', 'p'))
False
```

Returns boolean.

Special methods

`(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)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`(Spanner).__getitem__(expr)`
 Gets item from spanner.
 Returns component.

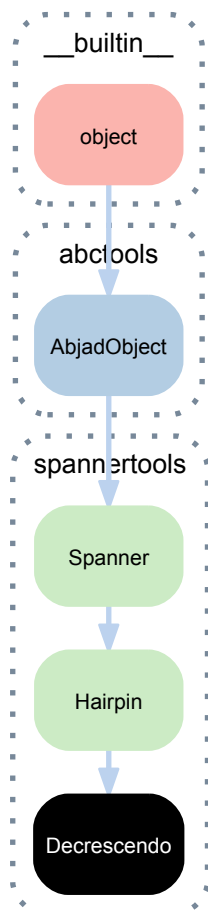
`(Spanner).__len__()`
 Length of spanner.
 Returns nonnegative integer.

`(Spanner).__lt__(expr)`
 True when spanner is less than *expr*.
 Trivial comparison to allow doctests to work.
 Returns boolean.

(AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

22.1.4 spannertools.Decrescendo



class spannertools.**Decrescendo** (*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")
>>> show(staff)
```



```
>>> decrescendo = spannertools.Decrescendo(include_rests=True)
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



Abjad decrescendo spanner that does not include rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> show(staff)
```



```
>>> decrescendo = spannertools.Decrescendo(include_rests=False)
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



Bases

- `spannertools.Hairpin`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**
Components in spanner.

Returns tuple.

(Spanner) **.leaves**
Leaves in spanner.

Returns tuple.

Read/write properties

(Hairpin) **.direction**
Gets and sets direction of hairpin.

Returns up or down.

(Hairpin) **.include_rests**
Gets and sets boolean setting to include rests in hairpin.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     include_rests=True,
... )
>>> attach(hairpin, staff[:])
>>> hairpin.include_rests
True
```

Sets include-rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     include_rests=True,
... )
>>> attach(hairpin, staff[:])
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Returns boolean.

`(Hairpin).shape_string`

Gets and sets hairpin shape string.

Gets hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.shape_string
'<'
```

Sets hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.shape_string = '>'
>>> hairpin.shape_string
'>'
```

Returns string.

`(Hairpin).start_dynamic_string`

Gets and sets start dynamic string of hairpin.

Gets start dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.start_dynamic_string
'p'
```

Sets start dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Returns string.

`(Hairpin).stop_dynamic_string`

Gets and sets stop dynamic string of hairpin.

Gets stop dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.stop_dynamic_string
'f'
```

Sets stop dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Returns string.

Methods

- (`Spanner`) .**append** (*component*)
 Appends *component* to spanner.
 Returns none.
- (`Spanner`) .**append_left** (*component*)
 Appends *component* to left of spanner.
 Returns none.
- (`Spanner`) .**detach** ()
 Detaches spanner.
 Returns none.
- (`Spanner`) .**extend** (*components*)
 Extends spanner with *components*.
 Returns none.
- (`Spanner`) .**extend_left** (*components*)
 Extends left of spanner with *components*.
 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`.
 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*.
 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.
 Returns nonnegative integer.
- (`Spanner`) .**pop** ()
 Pops rightmost component off of spanner.
 Returns component.
- (`Spanner`) .**pop_left** ()
 Pops leftmost component off of spanner.
 Returns component.

Static methods

`(Hairpin).is_hairpin_shape_string(arg)`
True when *arg* is a hairpin shape string. Otherwise false:

```
>>> spannertools.Hairpin.is_hairpin_shape_string('<')
True
```

Returns boolean.

`(Hairpin).is_hairpin_token(arg)`
True when *arg* is a hairpin token. Otherwise false:

```
>>> spannertools.Hairpin.is_hairpin_token(('p', '<', 'f'))
True
```

```
>>> spannertools.Hairpin.is_hairpin_token(('f', '<', 'p'))
False
```

Returns boolean.

Special methods

`(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)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(Spanner).__getitem__(expr)`
Gets item from spanner.
Returns component.

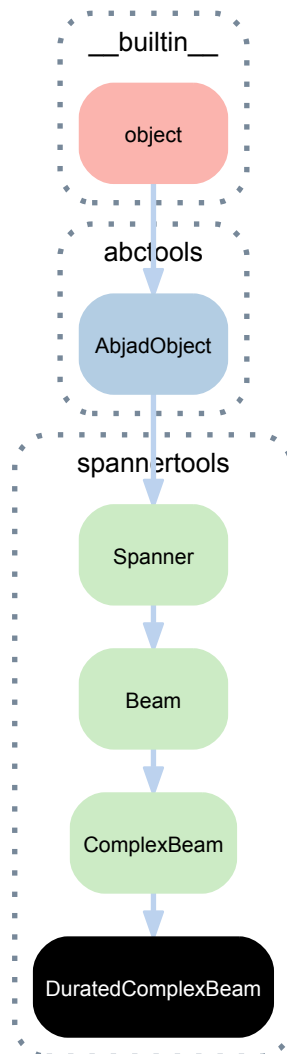
`(Spanner).__len__()`
Length of spanner.
Returns nonnegative integer.

`(Spanner).__lt__(expr)`
True when spanner is less than *expr*.
Trivial comparison to allow doctests to work.
Returns boolean.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

22.1.5 spannertools.DuratedComplexBeam



class spannertools.**DuratedComplexBeam**(*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.DuratedComplexBeam(
...     durations=durations,
...     span=1,
... )
>>> attach(beam, staff[:])
>>> 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.ComplexBeam`
- `spannertools.Beam`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**
 Components in spanner.
 Returns tuple.

(Spanner) **.leaves**
 Leaves in spanner.
 Returns tuple.

Read/write properties

(Beam) **.direction**
 Gets and sets direction of beam.
 Returns up or down.

DuratedComplexBeam **.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.DuratedComplexBeam(
...     durations=durations,
... )
>>> attach(beam, staff[:])
>>> 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.DuratedComplexBeam(
...     durations=durations,
... )
>>> attach(beam, staff[:])
>>> beam.durations = [Duration(1, 4)]
>>> beam.durations
[Duration(1, 4)]
```

Set iterable.

(ComplexBeam) **.lone**
 Beam lone leaf and force beam nibs to left:


```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='left')
>>> attach(beam, note)
>>> show(note)
```



Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='right')
>>> attach(beam, note)
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='both')
>>> attach(beam, note)
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone=True)
>>> attach(beam, note)
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone=False)
>>> attach(beam, note)
```

Set to 'left', 'right', 'both', true or false as shown above.

Ignore this setting when spanner contains more than one leaf.

`DuratedComplexBeam`. **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.DuratedComplexBeam(
...     durations=durations,
...     span=1,
... )
>>> attach(beam, staff[:])
>>> 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.DuratedComplexBeam(
...     durations=durations,
...     span=1,
... )
>>> attach(beam, staff[:])
>>> beam.span = 2
>>> beam.span
2
```

Set nonnegative integer.

Methods

- (`Spanner`) .**append** (*component*)
Appends *component* to spanner.
Returns none.
- (`Spanner`) .**append_left** (*component*)
Appends *component* to left of spanner.
Returns none.
- (`Spanner`) .**detach** ()
Detaches spanner.
Returns none.
- (`Spanner`) .**extend** (*components*)
Extends spanner with *components*.
Returns none.
- (`Spanner`) .**extend_left** (*components*)
Extends left of spanner with *components*.
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`.
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*.
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.
Returns nonnegative integer.
- (`Spanner`) .**pop** ()
Pops rightmost component off of spanner.
Returns component.
- (`Spanner`) .**pop_left** ()
Pops leftmost component off of spanner.
Returns component.

Static methods

(Beam) **.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.Beam
...     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) **.__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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) **.__format__** (*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

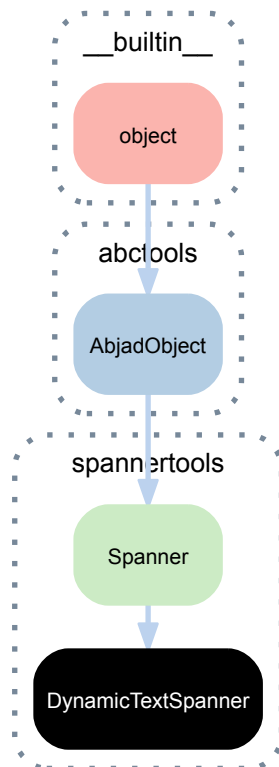
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

22.1.6 spannertools.DynamicTextSpanner



class spannertools.**DynamicTextSpanner** (*components=None, dynamic='', overrides=None*)

A dynamic text spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.DynamicTextSpanner(dynamic='f')
>>> attach(spanner, staff[:])
>>> show(staff)
```



Formats *dynamic* on first leaf in spanner.

Bases

- spannertools.Spanner
- abctools.AbjadObject
- __builtin__.object

Read-only properties

`(Spanner) .components`
Components in spanner.

Returns tuple.

`(Spanner) .leaves`
Leaves in spanner.

Returns tuple.

Read/write properties

`DynamicTextSpanner.dynamic`
Get dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.DynamicTextSpanner(dynamic='f')
>>> attach(spanner, staff[:])
>>> spanner.dynamic
'f'
```

Set dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.DynamicTextSpanner(dynamic='f')
>>> attach(spanner, staff[:])
>>> spanner.dynamic = 'p'
>>> spanner.dynamic
'p'
```

Set string.

Methods

`(Spanner) .append(component)`
Appends *component* to spanner.

Returns none.

`(Spanner) .append_left(component)`
Appends *component* to left of spanner.

Returns none.

`(Spanner) .detach()`
Detaches spanner.

Returns none.

`(Spanner) .extend(components)`
Extends spanner with *components*.

Returns none.

`(Spanner) .extend_left(components)`
Extends left of spanner with *components*.

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.

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*.

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.

Returns nonnegative integer.

(*Spanner*) . **pop** ()

Pops rightmost component off of spanner.

Returns component.

(*Spanner*) . **pop_left** ()

Pops leftmost component off of spanner.

Returns component.

Special methods

(*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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(*AbjadObject*) . **__format__** (*format_specification=''*)

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(*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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

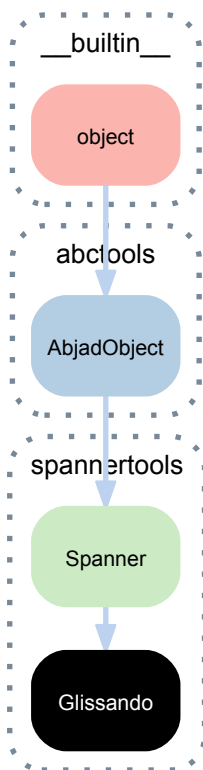
Returns boolean.

(AbjadObject) .__repr__()

Gets interpreter representation of Abjad object.

Returns string.

22.1.7 spannertools.Glissando



class spannertools.**Glissando** (*components=None, overrides=None, avoid_rests=False*)

A glissando spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> glissando = spannertools.Glissando()
>>> attach(glissando, staff[:])
>>> show(staff)
```



Can avoid rests:

```
>>> staff = Staff("c'16 r r g' r8 c'8")
>>> glissando = spannertools.Glissando(avoid_rests=True)
>>> attach(glissando, staff[:])
>>> beam = Beam()
>>> attach(beam, staff[:])
>>> show(staff)
```



Formats nonlast leaves in spanner with LilyPond glissando command.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(Spanner).components`

Components in spanner.

Returns tuple.

`(Spanner).leaves`

Leaves in spanner.

Returns tuple.

Read/write properties

`Glissando.avoid_rests`

Gets and sets rest avoidance.

Gets property:

```
>>> glissando.avoid_rests
True
```

Sets property:

```
>>> glissando.avoid_rests = False
>>> glissando.avoid_rests
False
```

Returns boolean.

Methods

`(Spanner).append(component)`

Appends *component* to spanner.

Returns none.

`(Spanner).append_left(component)`

Appends *component* to left of spanner.

Returns none.

`(Spanner).detach()`

Detaches spanner.

Returns none.

`(Spanner).extend(components)`

Extends spanner with *components*.

Returns none.

- (Spanner) **.extend_left** (*components*)
 Extends left of spanner with *components*.
 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.
 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*.
 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.
 Returns nonnegative integer.
- (Spanner) **.pop** ()
 Pops rightmost component off of spanner.
 Returns component.
- (Spanner) **.pop_left** ()
 Pops leftmost component off of spanner.
 Returns component.

Special methods

- (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*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.
- (AbjadObject) **.__format__** (*format_specification=''*)
 Formats object.
 Set *format_specification* to *'* or *'storage'*. Interprets *'* equal to *'storage'*.
 Returns string.

(*Spanner*) .__getitem__(*expr*)
 Gets item from spanner.
 Returns component.

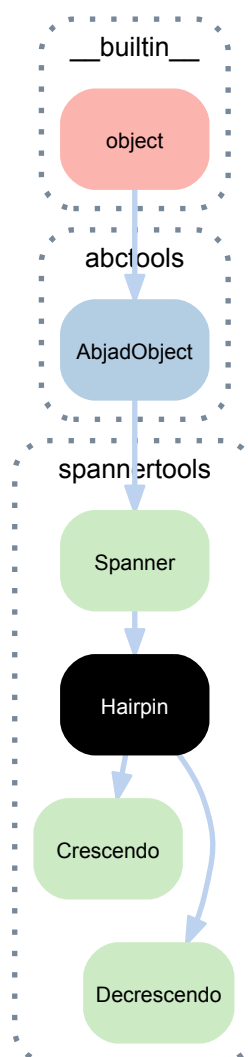
(*Spanner*) .__len__() *len*
 Length of spanner.
 Returns nonnegative integer.

(*Spanner*) .__lt__(*expr*) *lt*
 True when spanner is less than *expr*.
 Trivial comparison to allow doctests to work.
 Returns boolean.

(*AbjadObject*) .__ne__(*expr*) *ne*
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(*AbjadObject*) .__repr__() *repr*
 Gets interpreter representation of Abjad object.
 Returns string.

22.1.8 spannertools.Hairpin



class `spannertools.Hairpin` (*components=None*, *descriptor='<'*, *include_rests=True*, *direction=None*, *overrides=None*)

A hairpin.

Example 1. Hairpin spanner that does not include rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     include_rests=False,
... )
>>> attach(hairpin, staff[:])
>>> show(staff)
```



Example 2. Hairpin spanner that includes rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     include_rests=True,
... )
>>> attach(hairpin, staff[:])
>>> show(staff)
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
Components in spanner.

Returns tuple.

`(Spanner).leaves`
Leaves in spanner.

Returns tuple.

Read/write properties

`Hairpin.direction`
Gets and sets direction of hairpin.

Returns up or down.

`Hairpin.include_rests`
Gets and sets boolean setting to include rests in hairpin.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     include_rests=True,
```

```
...    )
>>> attach(hairpin, staff[:])
>>> hairpin.include_rests
True
```

Sets include-rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     include_rests=True,
... )
>>> attach(hairpin, staff[:])
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Returns boolean.

Hairpin.shape_string

Gets and sets hairpin shape string.

Gets hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.shape_string
'<'
```

Sets hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.shape_string = '>'
>>> hairpin.shape_string
'>'
```

Returns string.

Hairpin.start_dynamic_string

Gets and sets start dynamic string of hairpin.

Gets start dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.start_dynamic_string
'p'
```

Sets start dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Returns string.

Hairpin.stop_dynamic_string

Gets and sets stop dynamic string of hairpin.

Gets stop dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
```

```
>>> hairpin.stop_dynamic_string
'f'
```

Sets stop dynamic string of hairpin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Returns string.

Methods

(Spanner) . **append** (*component*)
 Appends *component* to spanner.

Returns none.

(Spanner) . **append_left** (*component*)
 Appends *component* to left of spanner.

Returns none.

(Spanner) . **detach** ()
 Detaches spanner.

Returns none.

(Spanner) . **extend** (*components*)
 Extends spanner with *components*.

Returns none.

(Spanner) . **extend_left** (*components*)
 Extends left of spanner with *components*.

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.

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*.

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.

Returns nonnegative integer.

`(Spanner) .pop()`
 Pops rightmost component off of spanner.
 Returns component.

`(Spanner) .pop_left()`
 Pops leftmost component off of spanner.
 Returns component.

Static methods

`Hairpin.is_hairpin_shape_string(arg)`
 True when *arg* is a hairpin shape string. Otherwise false:

```
>>> spannertools.Hairpin.is_hairpin_shape_string('<')
True
```

Returns boolean.

`Hairpin.is_hairpin_token(arg)`
 True when *arg* is a hairpin token. Otherwise false:

```
>>> spannertools.Hairpin.is_hairpin_token(('p', '<', 'f'))
True
```

```
>>> spannertools.Hairpin.is_hairpin_token(('f', '<', 'p'))
False
```

Returns boolean.

Special methods

`(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)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject) .__format__(format_specification='')`
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

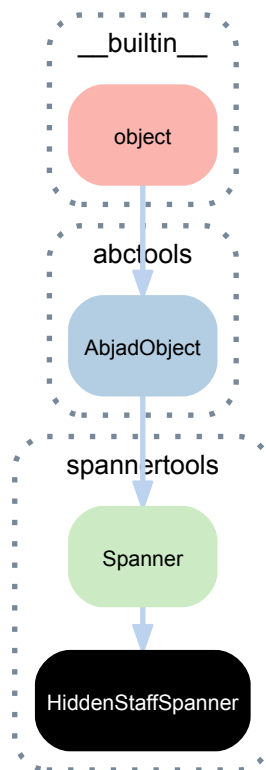
Returns boolean.

(AbjadObject) .__repr__()

Gets interpreter representation of Abjad object.

Returns string.

22.1.9 spannertools.HiddenStaffSpanner

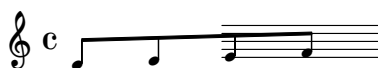


class spannertools.**HiddenStaffSpanner** (*components=None, overrides=None*)

A hidden staff spanner.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.HiddenStaffSpanner()
>>> attach(spanner, staff[:2])
>>> show(staff)
  
```



Hide staff behind leaves in spanner.

Bases

- spannertools.Spanner
- abctools.AbjadObject

- `__builtin__.object`

Read-only properties

`(Spanner) .components`
Components in spanner.
Returns tuple.

`(Spanner) .leaves`
Leaves in spanner.
Returns tuple.

Methods

`(Spanner) .append(component)`
Appends *component* to spanner.
Returns none.

`(Spanner) .append_left(component)`
Appends *component* to left of spanner.
Returns none.

`(Spanner) .detach()`
Detaches spanner.
Returns none.

`(Spanner) .extend(components)`
Extends spanner with *components*.
Returns none.

`(Spanner) .extend_left(components)`
Extends left of spanner with *components*.
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.
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*.
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.
Returns nonnegative integer.

`(Spanner).pop()`
Pops rightmost component off of spanner.
Returns component.

`(Spanner).pop_left()`
Pops leftmost component off of spanner.
Returns component.

Special methods

`(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)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(Spanner).__getitem__(expr)`
Gets item from spanner.
Returns component.

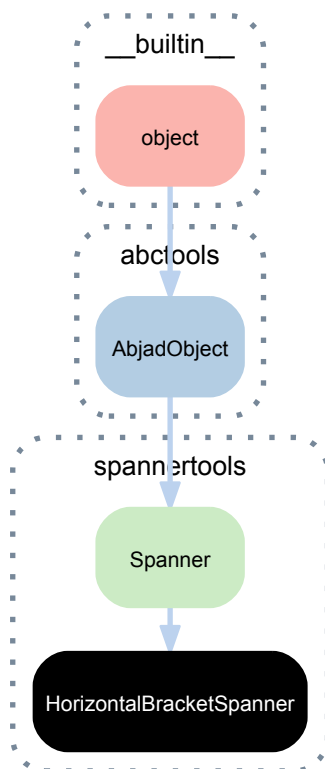
`(Spanner).__len__()`
Length of spanner.
Returns nonnegative integer.

`(Spanner).__lt__(expr)`
True when spanner is less than *expr*.
Trivial comparison to allow doctests to work.
Returns boolean.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

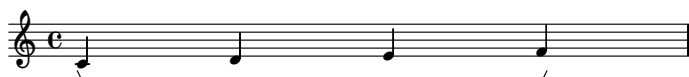
`(AbjadObject).__repr__()`
Gets interpreter representation of Abjad object.
Returns string.

22.1.10 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')
>>> spanner = spannertools.HorizontalBracketSpanner()
>>> attach(spanner, voice[:])
>>> show(voice)
```



```
>>> spanner
HorizontalBracketSpanner("c'4, d'4, e'4, f'4")
```

Bases

- spannertools.Spanner
- abctools.AbjadObject
- __builtin__.object

Read-only properties

(Spanner).**components**
 Components in spanner.

Returns tuple.

(Spanner).**leaves**
 Leaves in spanner.

Returns tuple.

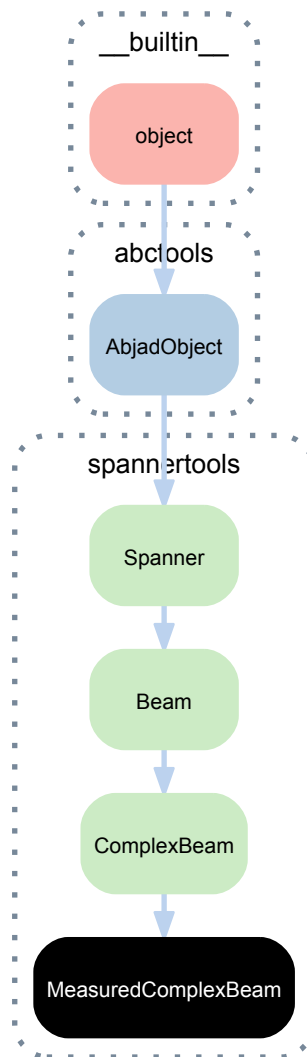
Methods

- (`Spanner`) **.append** (*component*)
 Appends *component* to spanner.
 Returns none.
- (`Spanner`) **.append_left** (*component*)
 Appends *component* to left of spanner.
 Returns none.
- (`Spanner`) **.detach** ()
 Detaches spanner.
 Returns none.
- (`Spanner`) **.extend** (*components*)
 Extends spanner with *components*.
 Returns none.
- (`Spanner`) **.extend_left** (*components*)
 Extends left of spanner with *components*.
 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`.
 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*.
 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.
 Returns nonnegative integer.
- (`Spanner`) **.pop** ()
 Pops rightmost component off of spanner.
 Returns component.
- (`Spanner`) **.pop_left** ()
 Pops leftmost component off of spanner.
 Returns component.

Special methods

- (*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*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.
- (*AbjadObject*) .__format__ (*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.
- (*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*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.
- (*AbjadObject*) .__repr__ ()
 Gets interpreter representation of Abjad object.
 Returns string.

22.1.11 spannertools.MeasuredComplexBeam



class `spannertools.MeasuredComplexBeam`(*components=None, lone=False, span=1, direction=None, overrides=None*)

A measured complex beam spanner.

```

>>> staff = Staff()
>>> staff.append(Measure((2, 16), "c'16 d'16"))
>>> staff.append(Measure((2, 16), "e'16 f'16"))
>>> show(staff)
  
```



```

>>> beam = spannertools.MeasuredComplexBeam()
>>> attach(beam, staff.select_leaves())
>>> show(staff)
  
```



Beams leaves in spanner explicitly.

Groups leaves by measures.

Formats top-level *span* beam between measures.

Bases

- `spannertools.ComplexBeam`
- `spannertools.Beam`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**

Components in spanner.

Returns tuple.

(Spanner) **.leaves**

Leaves in spanner.

Returns tuple.

Read/write properties

(Beam) **.direction**

Gets and sets direction of beam.

Returns up or down.

(ComplexBeam) **.lone**

Beam lone leaf and force beam nibs to left:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='left')
>>> attach(beam, note)
>>> show(note)
```



Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='right')
>>> attach(beam, note)
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone='both')
>>> attach(beam, note)
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone=True)
>>> attach(beam, note)
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = spannertools.ComplexBeam(lone=False)
>>> attach(beam, note)
```

Set to 'left', 'right', 'both', true or false as shown above.

Ignore this setting when spanner contains more than one leaf.

`MeasuredComplexBeam`. **span**

Get top-level beam count:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 16), "c'16 d'16"))
>>> staff.append(Measure((2, 16), "e'16 f'16"))
>>> beam = spannertools.MeasuredComplexBeam()
>>> attach(beam, staff.select_leaves())
>>> beam.span
1
```

Set top-level beam count:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 16), "c'16 d'16"))
>>> staff.append(Measure((2, 16), "e'16 f'16"))
>>> beam = spannertools.MeasuredComplexBeam()
>>> attach(beam, staff.select_leaves())
>>> beam.span = 2
>>> beam.span
2
```

Set nonnegative integer.

Methods

`(Spanner)` . **append** (*component*)
Appends *component* to spanner.

Returns none.

`(Spanner)` . **append_left** (*component*)
Appends *component* to left of spanner.

Returns none.

`(Spanner)` . **detach** ()
Detaches spanner.

Returns none.

`(Spanner)` . **extend** (*components*)
Extends spanner with *components*.

Returns none.

`(Spanner)` . **extend_left** (*components*)
Extends left of spanner with *components*.

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.

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*.
 - 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**.
 - Returns nonnegative integer.
- (**Spanner**) **.pop** ()
 - Pops rightmost component off of **spanner**.
 - Returns component.
- (**Spanner**) **.pop_left** ()
 - Pops leftmost component off of **spanner**.
 - Returns component.

Static methods

(Beam) **.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.Beam
...     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).__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*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

(*AbjadObject*) .__**format**__ (*format_specification*='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(*Spanner*) .__**getitem**__ (*expr*)
Gets item from spanner.
Returns component.

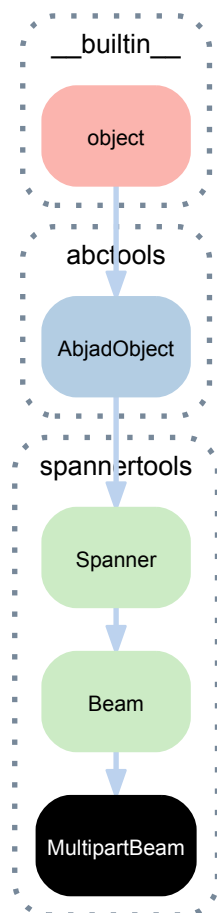
(*Spanner*) .__**len**__ ()
Length of spanner.
Returns nonnegative integer.

(*Spanner*) .__**lt**__ (*expr*)
True when spanner is less than *expr*.
Trivial comparison to allow doctests to work.
Returns boolean.

(*AbjadObject*) .__**ne**__ (*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

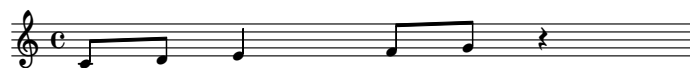
(*AbjadObject*) .__**repr**__ ()
Gets interpreter representation of Abjad object.
Returns string.

22.1.12 spannertools.MultipartBeam

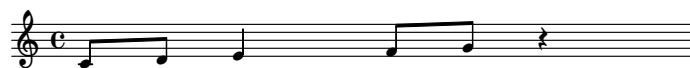


class `spannertools.MultipartBeam` (*components=None, direction=None, overrides=None*)
 A multipart beam spanner.

```
>>> staff = Staff("c'8 d'8 e'4 f'8 g'8 r4")
>>> show(staff)
```



```
>>> beam = spannertools.MultipartBeam()
>>> attach(beam, staff[:])
>>> show(staff)
```



Avoids rests.

Avoids large-duration notes.

Bases

- `spannertools.Beam`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(`Spanner`) .**components**
Components in spanner.

Returns tuple.

(`Spanner`) .**leaves**
Leaves in spanner.

Returns tuple.

Read/write properties

(`Beam`) .**direction**
Gets and sets direction of beam.

Returns up or down.

Methods

(`Spanner`) .**append** (*component*)
Appends *component* to spanner.

Returns none.

(`Spanner`) .**append_left** (*component*)
Appends *component* to left of spanner.

Returns none.

(`Spanner`) .**detach** ()
Detaches spanner.

Returns none.

(`Spanner`) .**extend** (*components*)
Extends spanner with *components*.

Returns none.

(`Spanner`) .**extend_left** (*components*)
Extends left of spanner with *components*.

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`.
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*.
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.

Returns nonnegative integer.

(Spanner) **.pop** ()

Pops rightmost component off of spanner.

Returns component.

(Spanner) **.pop_left** ()

Pops leftmost component off of spanner.

Returns component.

Static methods

(Beam) **.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.Beam
...     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) **.__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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(Spanner).**__getitem__**(*expr*)
 Gets item from spanner.
 Returns component.

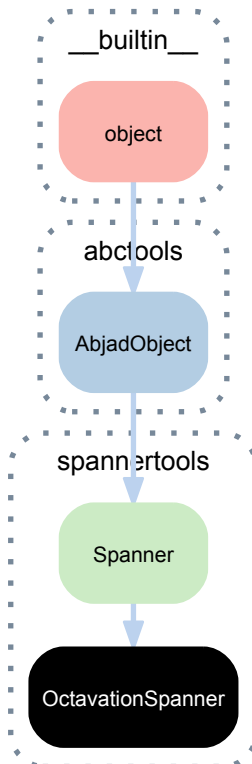
(Spanner).**__len__**()
 Length of spanner.
 Returns nonnegative integer.

(Spanner).**__lt__**(*expr*)
 True when spanner is less than *expr*.
 Trivial comparison to allow doctests to work.
 Returns boolean.

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

22.1.13 spannertools.OctavationSpanner

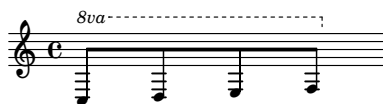


class spannertools.**OctavationSpanner**(*components=None, start=1, stop=0, overrides=None*)
 An octavation spanner.

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



```
>>> spanner = spannertools.OctavationSpanner(start=1)
>>> attach(spanner, staff[:])
>>> show(staff)
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
Components in spanner.

Returns tuple.

`(Spanner).leaves`
Leaves in spanner.

Returns tuple.

Read/write properties

`OctavationSpanner.start`
Get octavation start:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.OctavationSpanner(start=1)
>>> attach(spanner, staff[:])
>>> spanner.start
1
```

Set octavation start:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.OctavationSpanner(start=1)
>>> attach(spanner, staff[:])
>>> spanner.start
1
```

Set integer.

`OctavationSpanner.stop`
Get octavation stop:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.OctavationSpanner(start=2, stop=1)
>>> attach(spanner, staff[:])
>>> spanner.stop
1
```

Set octavation stop:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.OctavationSpanner(start=2, stop=1)
>>> attach(spanner, staff[:])
>>> spanner.stop = 0
>>> spanner.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()
>>> attach(octavation, 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.

Returns none.

(Spanner) .**append_left** (*component*)
Appends *component* to left of spanner.

Returns none.

(Spanner) .**detach** ()
Detaches spanner.

Returns none.

(Spanner) .**extend** (*components*)
Extends spanner with *components*.

Returns none.

(Spanner) .**extend_left** (*components*)
Extends left of spanner with *components*.

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.

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*.
 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*.
 Returns nonnegative integer.

(*Spanner*) . **pop** ()
 Pops rightmost component off of *spanner*.
 Returns component.

(*Spanner*) . **pop_left** ()
 Pops leftmost component off of *spanner*.
 Returns component.

Special methods

(*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*)
 Is true when ID of *expr* equals ID of *Abjad* object. Otherwise false.
 Returns boolean.

(*AbjadObject*) . **__format__** (*format_specification=''*)
 Formats object.
 Set *format_specification* to *'* or *'storage'*. Interprets *'* equal to *'storage'*.
 Returns string.

(*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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

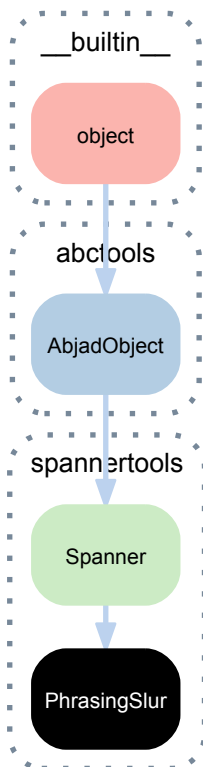
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

22.1.14 spannertools.PhrasingSlur



class spannertools.**PhrasingSlur** (*components=None, direction=None, overrides=None*)

A phrasing slur.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.PhrasingSlur()
>>> attach(slur, staff[:])
>>> show(staff)
```



Bases

- spannertools.Spanner
- abctools.AbjadObject
- __builtin__.object

Read-only properties

(Spanner) .**components**
Components in spanner.

Returns tuple.

(Spanner) .**leaves**
Leaves in spanner.

Returns tuple.

Read/write properties

PhrasingSlur.**direction**
Gets and sets direction of phrasing slur.

Returns up or down.

Methods

(Spanner) .**append** (*component*)
Appends *component* to spanner.

Returns none.

(Spanner) .**append_left** (*component*)
Appends *component* to left of spanner.

Returns none.

(Spanner) .**detach** ()
Detaches spanner.

Returns none.

(Spanner) .**extend** (*components*)
Extends spanner with *components*.

Returns none.

(Spanner) .**extend_left** (*components*)
Extends left of spanner with *components*.

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.
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*.
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.

Returns nonnegative integer.

(Spanner) .**pop** ()

Pops rightmost component off of spanner.

Returns component.

(Spanner) .**pop_left** ()

Pops leftmost component off of spanner.

Returns component.

Special methods

(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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__** (*format_specification=''*)

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(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.

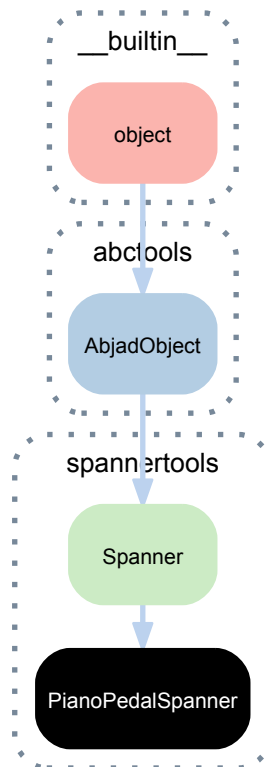
(AbjadObject) .**__ne__** (*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

22.1.15 spannertools.PianoPedalSpanner



class `spannertools.PianoPedalSpanner` (*components=None, overrides=None*)
 A piano pedal spanner.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> pedal = spannertools.PianoPedalSpanner()
>>> attach(pedal, staff[:])
>>> show(staff)
  
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
 Components in spanner.
 Returns tuple.

(Spanner) . **leaves**

Leaves in spanner.

Returns tuple.

Read/write properties

PianoPedalSpanner. **kind**

Get piano pedal spanner kind:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner()
>>> attach(spanner, staff[:])
>>> spanner.kind
'sustain'
```

Set piano pedal spanner kind:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner()
>>> attach(spanner, 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()
>>> attach(spanner, staff[:])
>>> spanner.style
'mixed'
```

Set piano pedal spanner style:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner()
>>> attach(spanner, staff[:])
>>> spanner.style = 'bracket'
>>> spanner.style
'bracket'
```

Acceptable values 'mixed', 'bracket', 'text'.

Methods

(Spanner) . **append** (*component*)

Appends *component* to spanner.

Returns none.

(Spanner) . **append_left** (*component*)

Appends *component* to left of spanner.

Returns none.

(Spanner) . **detach** ()

Detaches spanner.

Returns none.

(Spanner) . **extend** (*components*)

Extends spanner with *components*.

Returns none.

- (*Spanner*) .**extend_left** (*components*)
 Extends left of spanner with *components*.
 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`.
 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*.
 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.
 Returns nonnegative integer.
- (*Spanner*) .**pop** ()
 Pops rightmost component off of spanner.
 Returns component.
- (*Spanner*) .**pop_left** ()
 Pops leftmost component off of spanner.
 Returns component.

Special methods

- (*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*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.
- (*AbjadObject*) .**__format__** (*format_specification=''*)
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

(*Spanner*) . **__getitem__** (*expr*)
 Gets item from spanner.
 Returns component.

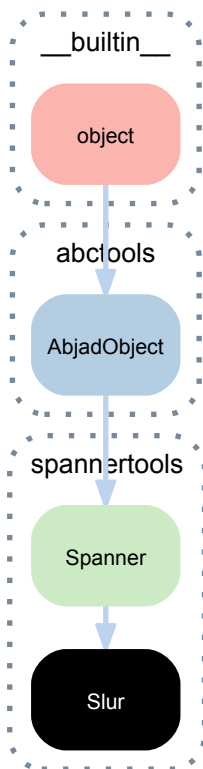
(*Spanner*) . **__len__** ()
 Length of spanner.
 Returns nonnegative integer.

(*Spanner*) . **__lt__** (*expr*)
 True when spanner is less than *expr*.
 Trivial comparison to allow doctests to work.
 Returns boolean.

(*AbjadObject*) . **__ne__** (*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(*AbjadObject*) . **__repr__** ()
 Gets interpreter representation of Abjad object.
 Returns string.

22.1.16 spannertools.Slur



class spannertools.**Slur** (*components=None, direction=None, overrides=None*)
 A slur.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.Slur()
>>> attach(slur, staff[:])
>>> show(staff)
  
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(`Spanner`) . **components**

Components in spanner.

Returns tuple.

(`Spanner`) . **leaves**

Leaves in spanner.

Returns tuple.

Read/write properties

`Slur` . **direction**

Gets and sets direction of slur.

Returns up, down or none.

Methods

(`Spanner`) . **append** (*component*)

Appends *component* to spanner.

Returns none.

(`Spanner`) . **append_left** (*component*)

Appends *component* to left of spanner.

Returns none.

(`Spanner`) . **detach** ()

Detaches spanner.

Returns none.

(`Spanner`) . **extend** (*components*)

Extends spanner with *components*.

Returns none.

(`Spanner`) . **extend_left** (*components*)

Extends left of spanner with *components*.

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.

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*.

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*.

Returns nonnegative integer.

(*Spanner*) . **pop** ()

Pops rightmost component off of *spanner*.

Returns component.

(*Spanner*) . **pop_left** ()

Pops leftmost component off of *spanner*.

Returns component.

Special methods

(*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*)

Is true when ID of *expr* equals ID of *Abjad* object. Otherwise false.

Returns boolean.

(*AbjadObject*) . **__format__** (*format_specification=''*)

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(*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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

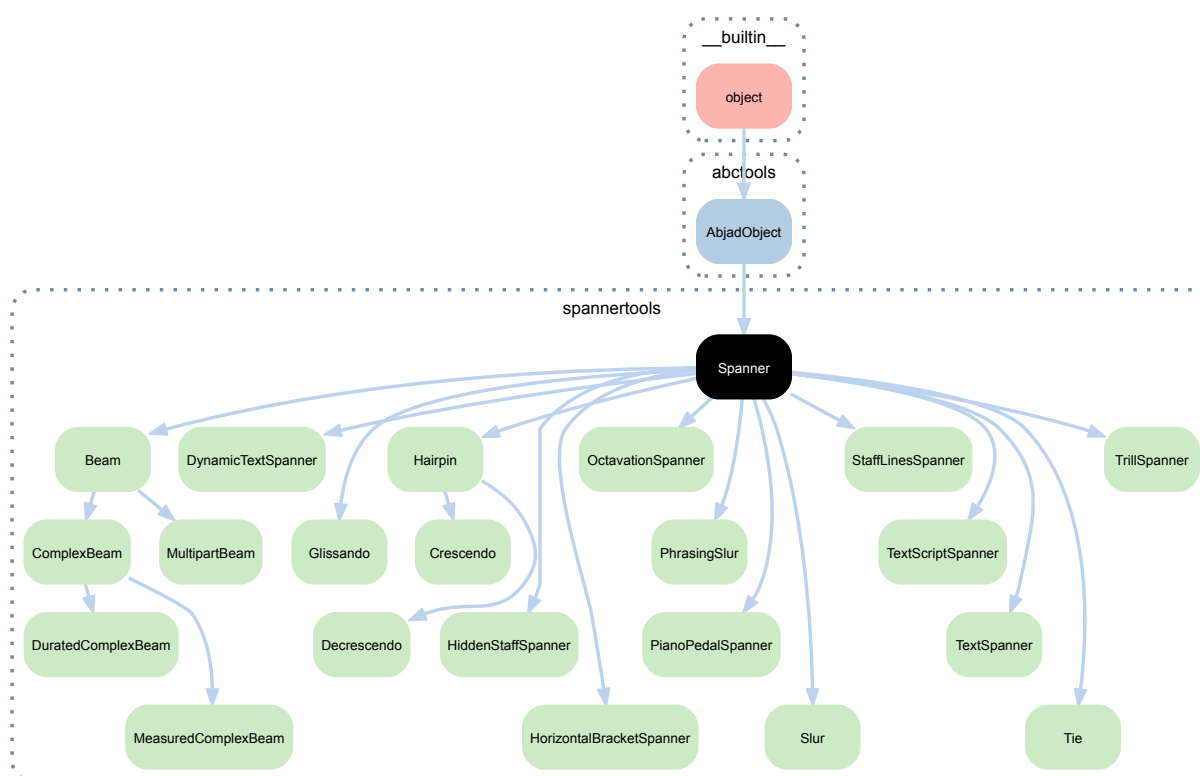
Returns boolean.

(AbjadObject) .__repr__()

Gets interpreter representation of Abjad object.

Returns string.

22.1.17 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 or other components. Examples include beams, slurs, hairpins and trills.

Bases

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

Read-only properties

Spanner.components

Components in spanner.

Returns tuple.

`Spanner.leaves`

Leaves in spanner.

Returns tuple.

Methods

`Spanner.append(component)`

Appends *component* to spanner.

Returns none.

`Spanner.append_left(component)`

Appends *component* to left of spanner.

Returns none.

`Spanner.detach()`

Detaches spanner.

Returns none.

`Spanner.extend(components)`

Extends spanner with *components*.

Returns none.

`Spanner.extend_left(components)`

Extends left of spanner with *components*.

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`.

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*.

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.

Returns nonnegative integer.

`Spanner.pop()`

Pops rightmost component off of spanner.

Returns component.

`Spanner.pop_left()`

Pops leftmost component off of spanner.

Returns component.

Special methods

`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)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`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)`

Is true when Abjad object does not equal *expr*. Otherwise false.

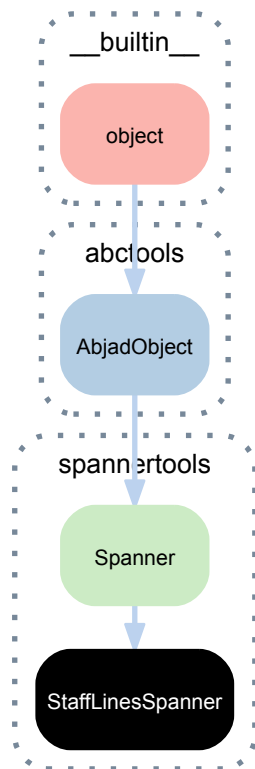
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

22.1.18 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")
>>> spanner = spannertools.StaffLinesSpanner(lines=1)
>>> attach(spanner, staff[:2])
>>> 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.

Returns tuple.

(`Spanner`) . **leaves**
Leaves in spanner.

Returns tuple.

Read/write properties

`StaffLinesSpanner.lines`

Get staff lines spanner line count:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.StaffLinesSpanner(lines=1)
>>> attach(spanner, staff[:2])
>>> spanner.lines
1
```

Set staff lines spanner line count:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.StaffLinesSpanner(lines=1)
>>> attach(spanner, staff[:2])
>>> spanner.lines = 2
>>> spanner.lines
2
```

Set integer.

Methods

`(Spanner) .append(component)`
 Appends *component* to spanner.

Returns none.

`(Spanner) .append_left(component)`
 Appends *component* to left of spanner.

Returns none.

`(Spanner) .detach()`
 Detaches spanner.

Returns none.

`(Spanner) .extend(components)`
 Extends spanner with *components*.

Returns none.

`(Spanner) .extend_left(components)`
 Extends left of spanner with *components*.

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`.

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*.

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.

Returns nonnegative integer.

(Spanner) .**pop** ()

Pops rightmost component off of spanner.

Returns component.

(Spanner) .**pop_left** ()

Pops leftmost component off of spanner.

Returns component.

Special methods

(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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__** (*format_specification=''*)

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(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.

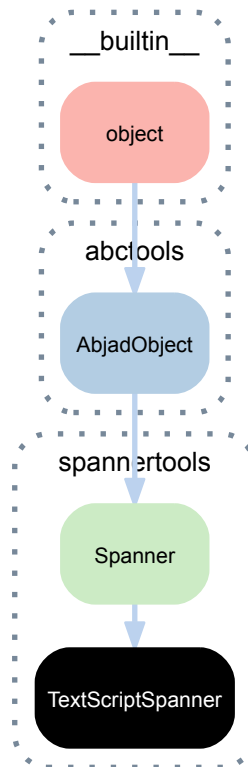
(AbjadObject) .**__ne__** (*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

22.1.19 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()
>>> attach(spanner, staff[:])
>>> override(spanner).text_script.color = 'red'
>>> markup = markuptools.Markup(r'\italic { espressivo }', Up)
>>> attach(markup, staff[1])
>>> show(staff)
  
```



Override LilyPond TextScript grob.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
 Components in spanner.

Returns tuple.

(`Spanner`) . **leaves**
Leaves in spanner.

Returns tuple.

Methods

(`Spanner`) . **append** (*component*)
Appends *component* to spanner.

Returns none.

(`Spanner`) . **append_left** (*component*)
Appends *component* to left of spanner.

Returns none.

(`Spanner`) . **detach** ()
Detaches spanner.

Returns none.

(`Spanner`) . **extend** (*components*)
Extends spanner with *components*.

Returns none.

(`Spanner`) . **extend_left** (*components*)
Extends left of spanner with *components*.

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`.
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*.
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.
Returns nonnegative integer.

(`Spanner`) . **pop** ()
Pops rightmost component off of spanner.
Returns component.

(*Spanner*) .**pop_left**()
Pops leftmost component off of spanner.
Returns component.

Special methods

(*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*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

(*AbjadObject*) .**__format__**(*format_specification*='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(*Spanner*) .**__getitem__**(*expr*)
Gets item from spanner.
Returns component.

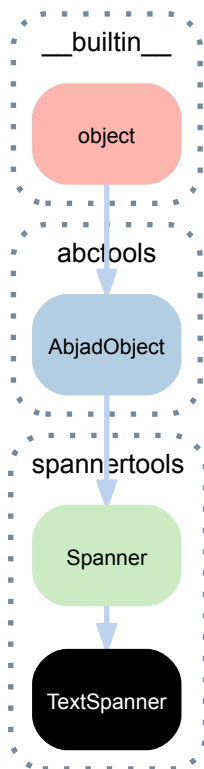
(*Spanner*) .**__len__**()
Length of spanner.
Returns nonnegative integer.

(*Spanner*) .**__lt__**(*expr*)
True when spanner is less than *expr*.
Trivial comparison to allow doctests to work.
Returns boolean.

(*AbjadObject*) .**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(*AbjadObject*) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

22.1.20 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 = override(text_spanner).text_spanner
>>> markup_command = markuptools.MarkupCommand('italic', 'foo')
>>> markup_command = markuptools.MarkupCommand('bold', markup_command)
>>> left_markup = markuptools.Markup(markup_command)
>>> grob.bind_details__left__text = left_markup
>>> pair = schemetools.SchemePair(0, -1)
>>> markup_command = markuptools.MarkupCommand('draw-line', pair)
>>> right_markup = markuptools.Markup(markup_command)
>>> grob.bind_details__right__text = right_markup
>>> override(text_spanner).text_spanner.dash_fraction = 1
>>> attach(text_spanner, [staff])
>>> show(staff)
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(*Spanner*) . **components**
 Components in spanner.

Returns tuple.

(*Spanner*) . **leaves**
 Leaves in spanner.

Returns tuple.

Methods

(*Spanner*) . **append** (*component*)
 Appends *component* to spanner.

Returns none.

(*Spanner*) . **append_left** (*component*)
 Appends *component* to left of spanner.

Returns none.

(*Spanner*) . **detach** ()
 Detaches spanner.

Returns none.

(*Spanner*) . **extend** (*components*)
 Extends spanner with *components*.

Returns none.

(*Spanner*) . **extend_left** (*components*)
 Extends left of spanner with *components*.

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`.
 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*.

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.

Returns nonnegative integer.

(*Spanner*).**pop**()
Pops rightmost component off of spanner.
Returns component.

(*Spanner*).**pop_left**()
Pops leftmost component off of spanner.
Returns component.

Special methods

(*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*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

(*AbjadObject*).**__format__**(*format_specification*='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(*Spanner*).**__getitem__**(*expr*)
Gets item from spanner.
Returns component.

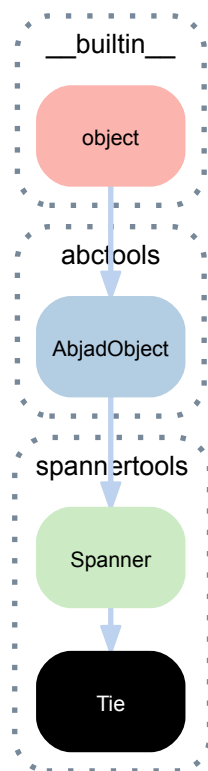
(*Spanner*).**__len__**()
Length of spanner.
Returns nonnegative integer.

(*Spanner*).**__lt__**(*expr*)
True when spanner is less than *expr*.
Trivial comparison to allow doctests to work.
Returns boolean.

(*AbjadObject*).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(*AbjadObject*).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

22.1.21 spannertools.Tie



class `spannertools.Tie` (*music=None, direction=None, overrides=None*)
 A tie.

```

>>> staff = Staff(scoretools.make_repeated_notes(4))
>>> tie = spannertools.Tie()
>>> attach(tie, staff[:])
>>> show(staff)
  
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(`Spanner`) **.components**
 Components in spanner.
 Returns tuple.

(`Spanner`) **.leaves**
 Leaves in spanner.
 Returns tuple.

Read/write properties

Tie.**direction**

Gets and sets direction of tie.

Returns up, down or none.

Methods

(Spanner) .**append** (*component*)

Appends *component* to spanner.

Returns none.

(Spanner) .**append_left** (*component*)

Appends *component* to left of spanner.

Returns none.

(Spanner) .**detach** ()

Detaches spanner.

Returns none.

(Spanner) .**extend** (*components*)

Extends spanner with *components*.

Returns none.

(Spanner) .**extend_left** (*components*)

Extends left of spanner with *components*.

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.

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*.

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.

Returns nonnegative integer.

(Spanner) .**pop** ()

Pops rightmost component off of spanner.

Returns component.

(*Spanner*) .**pop_left**()
Pops leftmost component off of spanner.
Returns component.

Special methods

(*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*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

(*AbjadObject*) .**__format__**(*format_specification*='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(*Spanner*) .**__getitem__**(*expr*)
Gets item from spanner.
Returns component.

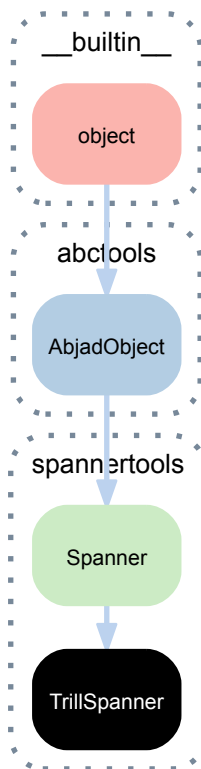
(*Spanner*) .**__len__**()
Length of spanner.
Returns nonnegative integer.

(*Spanner*) .**__lt__**(*expr*)
True when spanner is less than *expr*.
Trivial comparison to allow doctests to work.
Returns boolean.

(*AbjadObject*) .**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(*AbjadObject*) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

22.1.22 spannertools.TrillSpanner



class spannertools.**TrillSpanner** (*components=None, overrides=None*)
 A trill spanner.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> trill = spannertools.TrillSpanner()
>>> attach(trill, staff[:])
>>> show(staff)
  
```



Bases

- spannertools.Spanner
- abctools.AbjadObject
- __builtin__.object

Read-only properties

(Spanner).**components**
 Components in spanner.
 Returns tuple.

(Spanner).**leaves**
 Leaves in spanner.
 Returns tuple.

Read/write properties

`TrillSpanner.pitch`

Gets and sets optional pitch of trill spanner.

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> trill = spannertools.TrillSpanner()
>>> attach(trill, t[:2])
>>> trill.pitch = NamedPitch('cs', 4)
```

Returns pitch or none.

`TrillSpanner.written_pitch`

Gets and sets written pitch of trill spanner.

Returns pitch.

Methods

`(Spanner) .append(component)`

Appends *component* to spanner.

Returns none.

`(Spanner) .append_left(component)`

Appends *component* to left of spanner.

Returns none.

`(Spanner) .detach()`

Detaches spanner.

Returns none.

`(Spanner) .extend(components)`

Extends spanner with *components*.

Returns none.

`(Spanner) .extend_left(components)`

Extends left of spanner with *components*.

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.

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*.

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.
Returns nonnegative integer.
- (*Spanner*) . **pop** ()
Pops rightmost component off of spanner.
Returns component.
- (*Spanner*) . **pop_left** ()
Pops leftmost component off of spanner.
Returns component.

Special methods

- (*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*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.
- (*AbjadObject*) . **__format__** (*format_specification*='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.
- (*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*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.
- (*AbjadObject*) . **__repr__** ()
Gets interpreter representation of Abjad object.
Returns string.

22.2 Functions

22.2.1 `spannertools.make_colored_text_spanner_with_nibs`

`spannertools.make_colored_text_spanner_with_nibs()`

Makes colored text spanner with nibs.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.make_colored_text_spanner_with_nibs()
>>> attach(spanner, staff[:])
>>> show(staff)
```



Renders 1.5-unit thick solid red spanner.

Draws nibs at beginning and end of spanner.

Does not draw nibs at line breaks.

Returns bracket spanner.

22.2.2 `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.

22.2.3 `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.

23.1 Functions

23.1.1 `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.

23.1.2 `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.

23.1.3 `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.

23.1.4 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.

23.1.5 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.

23.1.6 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.

23.1.7 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.

23.1.8 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.

23.1.9 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")
... beam = spannertools.Beam()
... attach(beam, staff.select_leaves())
... f(staff)
...
... scoretools.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")
beam = spannertools.Beam()
attach(beam, staff.select_leaves())

r'''
\\new Staff {
  c'8 [
    d'8
    e'8
    f'8 ]
}
'''

scoretools.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 format(staff) == "\\new Staff {
  \\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.

23.1.10 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
```

Otherwise false:

```
>>> stringtools.is_dash_case_file_name('foo.bar.blah')
False
```

Returns boolean.

23.1.11 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
```

Otherwise false:

```
>>> stringtools.is_dash_case_string('foo bar')
False
```

Returns boolean.

23.1.12 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
```

Otherwise false:

```
>>> stringtools.is_lower_camel_case_string('FooBar')
False
```

Returns boolean.

23.1.13 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
```

Otherwise false:

```
>>> stringtools.is_snake_case_file_name('foo.bar.blah')
False
```

Returns boolean.

23.1.14 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
```

Otherwise false:

```
>>> stringtools.is_snake_case_file_name_with_extension('foo.bar.blah')
False
```

Returns boolean.

23.1.15 `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
```

Otherwise false:

```
>>> stringtools.is_snake_case_package_name('foo.bar.BlahPackage')
False
```

Returns boolean.

23.1.16 `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
```

Otherwise false:

```
>>> stringtools.is_snake_case_string('foo bar')
False
```

Returns boolean.

23.1.17 `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
```

Otherwise false:

```
>>> stringtools.is_space_delimited_lowercase_string('foo_bar')
False
```

Returns boolean.

23.1.18 `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
```

Otherwise false:

```
>>> stringtools.is_upper_camel_case_string('fooBar')
False
```

Returns boolean.

23.1.19 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.

23.1.20 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.

23.1.21 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.

23.1.22 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.

23.1.23 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('Déjà vu')
'deja_vu'
```

Strip accents from accented characters. Change all punctuation (including spaces) to underscore. Set to lowercase.

Returns string.

23.1.24 `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('LogicalTie')
'logical tie'
```

Change underscore-delimited *string* to space-delimited lowercase:

```
>>> stringtools.string_to_space_delimited_lowercase('logical_tie')
'logical tie'
```

Returns space-delimited string unchanged:

```
>>> stringtools.string_to_space_delimited_lowercase('logical tie')
'logical tie'
```

Returns empty *string* unchanged:

```
>>> stringtools.string_to_space_delimited_lowercase('')
''
```

Returns string.

23.1.25 `stringtools.strip_diacritics_from_binary_string`

`stringtools.strip_diacritics_from_binary_string` (*binary_string*)

Strip diacritics from *binary_string*:

```
>>> binary_string = 'Dvo\x99\x99\x99\x99\x99'
```

```
>>> print binary_string
Dvořák
```

```
>>> stringtools.strip_diacritics_from_binary_string(binary_string)
'Dvorak'
```

Returns ASCII string.

23.1.26 `stringtools.upper_camel_case_to_snake_case`

`stringtools.upper_camel_case_to_snake_case` (*string*)

Change uppercamelcase *string* to underscore-delimited lowercase:

```
>>> string = 'KeySignature'
```

```
>>> stringtools.upper_camel_case_to_snake_case(string)
'key_signature'
```

Returns string.

23.1.27 `stringtools.upper_camel_case_to_space_delimited_lowercase`

`stringtools.upper_camel_case_to_space_delimited_lowercase` (*string*)

Change uppercamelcase *string* to space-delimited lowercase:

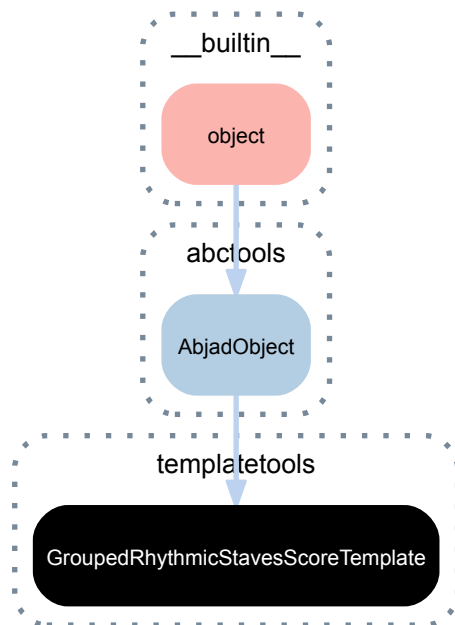
```
>>> string = 'KeySignature'
```

```
>>> stringtools.upper_camel_case_to_space_delimited_lowercase(string)
'key signature'
```

Returns string.

24.1 Concrete classes

24.1.1 `templatetools.GroupedRhythmicStavesScoreTemplate`



class `templatetools.GroupedRhythmicStavesScoreTemplate` (*staff_count=2*)
Grouped rhythmic staves score template.

```
>>> from abjad.tools.templatetools 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()
```

```
>>> print format(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()
```

```
>>> print format(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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

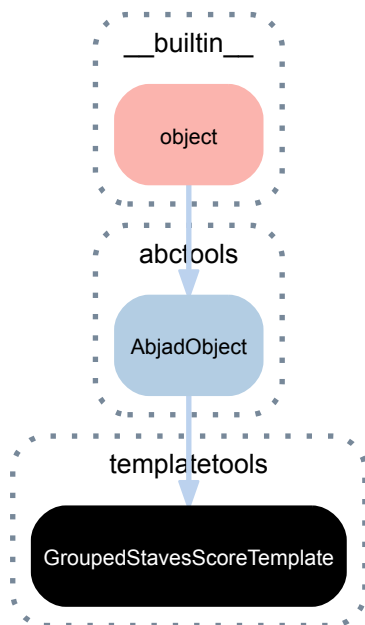
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

24.1.2 `templatetools.GroupedStavesScoreTemplate`



class `templatetools.GroupedStavesScoreTemplate` (*staff_count*=2)

Grouped staves score template.

```
>>> template_class = templatetools.GroupedStavesScoreTemplate
>>> template = template_class(staff_count=4)
```

Bases

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

Special methods

`GroupedStavesScoreTemplate.__call__()`

Calls score template.

```
>>> score = template()
```

```
>>> print format(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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

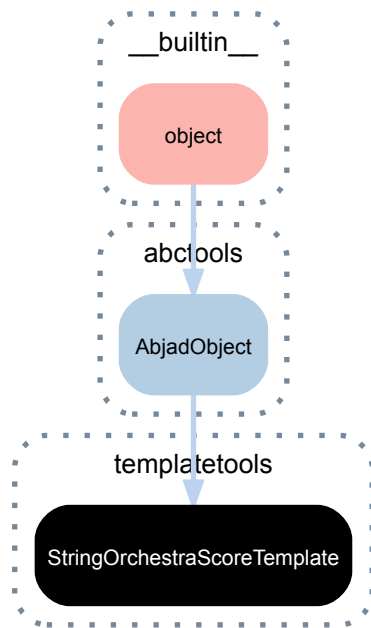
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

24.1.3 `templatetools.StringOrchestraScoreTemplate`



class `templatetools.StringOrchestraScoreTemplate` (*violin_count=6, viola_count=4, cello_count=3, contrabass_count=2*)

String orchestra score template.

```
>>> template = templatetools.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`
Number of cellos in string orchestra.

Returns nonnegative integer.

`StringOrchestraScoreTemplate.contrabass_count`
Number of contrabasses in string orchestra.

Returns nonnegative integer.

`StringOrchestraScoreTemplate.viola_count`
Number of violas in string orchestra.

Returns nonnegative integer.

`StringOrchestraScoreTemplate.violin_count`
 Number of violins in string orchestra.
 Returns nonnegative integer.

Special methods

`StringOrchestraScoreTemplate.__call__()`
 Calls string orchestra template.
 Returns score.

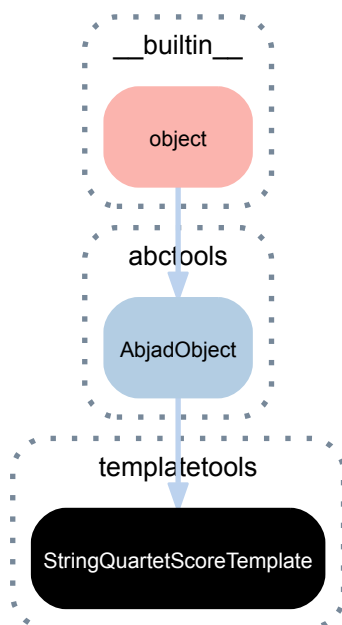
`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

24.1.4 `templatetools.StringQuartetScoreTemplate`



class `templatetools.StringQuartetScoreTemplate`
 String quartet score template.

```
>>> template = templatetools.StringQuartetScoreTemplate()
>>> score = template()
```

```
>>> score
Score="String Quartet Score"<<1>>
```

Returns score template.

Bases

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

Special methods

`StringQuartetScoreTemplate.__call__()`

Calls string quartet score template.

Returns score.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

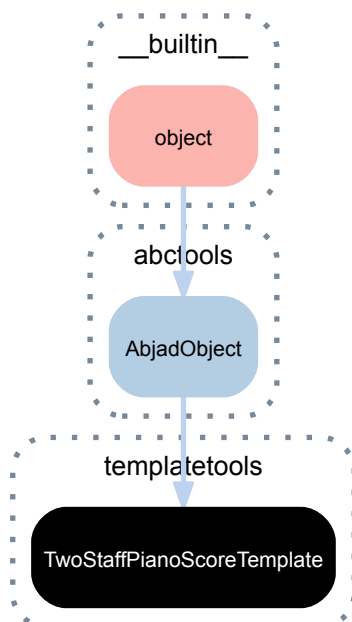
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

24.1.5 `templatetools.TwoStaffPianoScoreTemplate`



class `templatetools.TwoStaffPianoScoreTemplate`
Two-staff piano score template.

```
>>> template = templatetools.TwoStaffPianoScoreTemplate()  
>>> score = template()
```

```
>>> score  
Score-"Two-Staff Piano Score"<<1>>
```

Returns score template.

Bases

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

Special methods

`TwoStaffPianoScoreTemplate.__call__()`

Calls two-staff piano score template.

Returns score.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

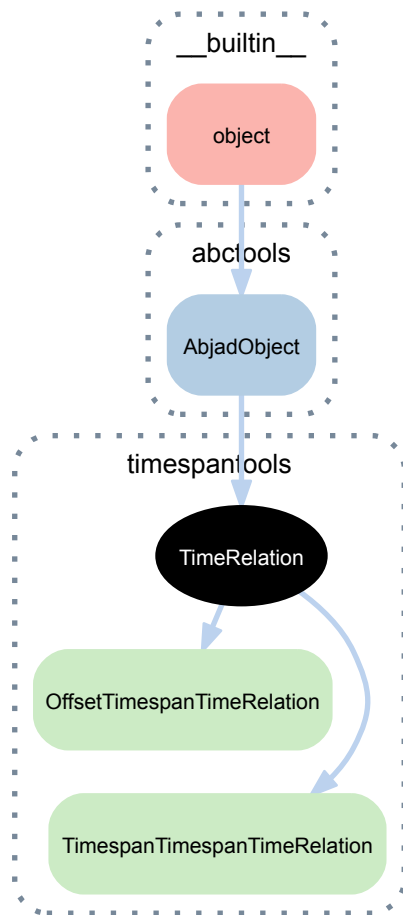
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

25.1 Abstract classes

25.1.1 timespantools.TimeRelation



class `timespantools.TimeRelation` (*inequality=None*)

A time relation.

Time relations are immutable.

Bases

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

Read-only properties

`TimeRelation.inequality`

Time relation inequality.

Return inequality.

`TimeRelation.is_fully_loaded`

True when both time relation terms are not none. Otherwise false:

Returns boolean.

`TimeRelation.is_fully_unloaded`

True when both time relation terms are none. Otherwise false:

Returns boolean.

Special methods

`TimeRelation.__call__()`

Evaluates time relation.

Returns boolean.

`TimeRelation.__eq__(expr)`

True when *expr* is a equal-valued time relation. Otherwise false.

Returns boolean.

`TimeRelation.__format__(format_specification='')`

Formats time relation.

Returns string.

`TimeRelation.__makenew__(*args, **kwargs)`

Makes new time relation with optional *args* and *kwargs*.

Returns new time relation.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

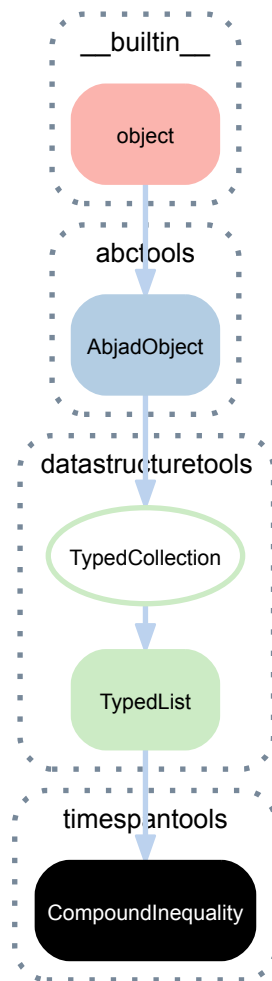
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

25.2 Concrete classes

25.2.1 timespantools.CompoundInequality



class `timespantools.CompoundInequality` (*tokens=None*, *logical_operator='and'*, *custom_identifier=None*)

A compound time-relation inequality.

```
>>> compound_inequality = timespantools.CompoundInequality([
...     timespantools.CompoundInequality([
...         'timespan_1.start_offset <= timespan_2.start_offset',
...         'timespan_2.start_offset < timespan_1.stop_offset'],
...         logical_operator='and'),
...     timespantools.CompoundInequality([
...         'timespan_2.start_offset <= timespan_1.start_offset',
...         'timespan_1.start_offset < timespan_2.stop_offset'],
...         logical_operator='and')],
...     logical_operator='or',
... )
```

```
>>> print format(compound_inequality)
timespantools.CompoundInequality(
[
    timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
            timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
        ],
        logical_operator='and',
    ),
    timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
            timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
        ],
        logical_operator='and',
    ),
],
logical_operator='or',
)
```

```
[
    timespantools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
    timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
],
    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.

Read/write properties

`(TypedCollection).custom_identifier`

Gets and sets custom identifier of typed collection.

Returns string or none.

`(TypedList).keep_sorted`

Sorts collection on mutation if true.

Methods

`(TypedList).append(token)`

Changes *token* to item and appends.

```
>>> 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)`

Changes *token* to item and returns 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*)

Evaluates compound inequality.

Returns boolean.

CompoundInequality.**evaluate_offset_inequality**(*timespan_start*, *timespan_stop*, *offset*)

Evaluates offset inequality.

Returns boolean.

(TypedList).**extend**(*tokens*)

Changes *tokens* to items and extends.

```
>>> 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*)

Gets offset indices of compound inequality.

(TypedList).**index**(*token*)

Changes *token* to item and returns index.

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

Returns index.

(TypedList).**insert**(*i*, *token*)

Changes *token* to item and inserts.

```
>>> 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.

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

Aliases list.pop().

(TypedList).**remove**(*token*)

Changes *token* to item and removes.

```
>>> 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)
True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList) .**__delitem__** (i)
Aliases list.__delitem__().

(TypedCollection) .**__eq__** (expr)
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection) .**__format__** (format_specification='')
Formats typed collection.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(TypedList) .**__getitem__** (i)
Aliases list.__getitem__().

(TypedList) .**__iadd__** (expr)
Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)  
datastructuretools.TypedList(  
    [  
        indicatortools.Dynamic(  
            'ppp'  
        ),  
        indicatortools.Dynamic(  
            'p'  
        ),  
        indicatortools.Dynamic(  
            'mp'  
        ),  
        indicatortools.Dynamic(  
            'mf'  
        ),  
        indicatortools.Dynamic(  
            'fff'
```

```

        ),
        ],
        item_class=indicatortools.Dynamic,
    )

```

Returns collection.

(TypedCollection) .**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList) .**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection) .**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList) .**__reversed__**()

Aliases list.__reversed__().

(TypedList) .**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

```

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

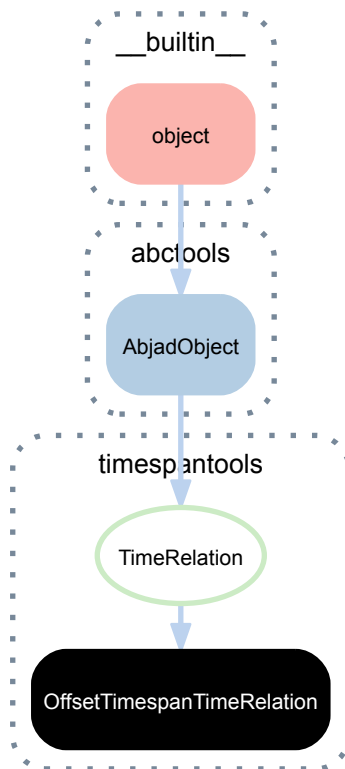
```

```

>>> pitch_collection[-1:] = ["f'", "g'", "a'", "b'", "c'"]
>>> print format(pitch_collection)
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,
)

```

25.2.2 timespantools.OffsetTimespanTimeRelation



class `timespantools.OffsetTimespanTimeRelation` (*inequality=None, timespan=None, offset=None*)

An offset vs. timespan time relation.

```
>>> offset = Offset(5)
>>> timespan = timespantools.Timespan(0, 10)
>>> time_relation = timespantools.offset_happens_during_timespan(
...     offset=offset,
...     timespan=timespan,
...     hold=True,
... )
```

```
>>> print format(time_relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan.start <= offset'),
            timespantools.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

- `timespantools.TimeRelation`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(TimeRelation).**inequality**

Time relation inequality.

Return 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.**timespan**

Time relation timespan:

```
>>> time_relation.timespan
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

Returns timespan or none.

Special methods

OffsetTimespanTimeRelation.**__call__** (*timespan=None, offset=None*)

Evaluates time relation:

```
>>> time_relation()
True
```

Raises value error is 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 = \
...     timespantools.offset_happens_during_timespan()
>>> time_relation_2 = \
...     timespantools.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.

`OffsetTimespanTimeRelation.__format__` (*format_specification*='')

Formats time relation.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

```
>>> print format(time_relation)
timespantools.OffsetTimespanTimeRelation(
  inequality=timespantools.CompoundInequality(
    [
      timespantools.SimpleInequality('timespan.start <= offset'),
      timespantools.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.

`(TimeRelation).__makenew__` (*args, **kwargs)

Makes new time relation with optional *args* and *kwargs*.

Returns new time relation.

`(AbjadObject).__ne__` (expr)

Is true when Abjad object does not equal *expr*. Otherwise false.

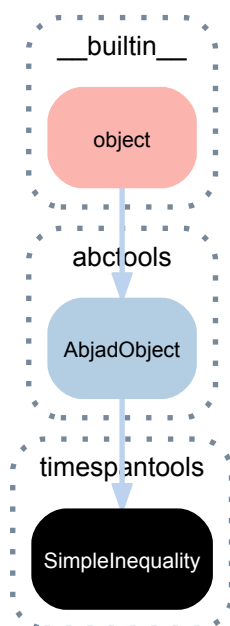
Returns boolean.

`(AbjadObject).__repr__` ()

Gets interpreter representation of Abjad object.

Returns string.

25.2.3 timespantools.SimpleInequality



class `timespantools.SimpleInequality` (*template=None*)

A simple inequality.

```
>>> template = 'timespan_2.start_offset < timespan_1.start_offset'
>>> simple_inequality = timespantools.SimpleInequality(template)
```

```
>>> simple_inequality
SimpleInequality('timespan_2.start_offset < timespan_1.start_offset')
```

Bases

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

Read-only properties

`SimpleInequality.template`
Template of simple inequality.

```
>>> 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*)

Evaluates simple inequality.

Returns boolean.

`SimpleInequality.evaluate_offset_inequality` (*timespan_start*, *timespan_stop*, *offset*)

Evaluates offset inequality.

Returns boolean.

`SimpleInequality.get_offset_indices` (*timespan_1*, *timespan_2_start_offsets*, *timespan_2_stop_offsets*)

Gets offset indices of simple inequality.

Todo

add example.

Returns nonnegative integer pair.

Special methods

(`AbjadObject`).`__eq__` (*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`SimpleInequality.__format__` (*format_specification*='')

Formats simple inequality.

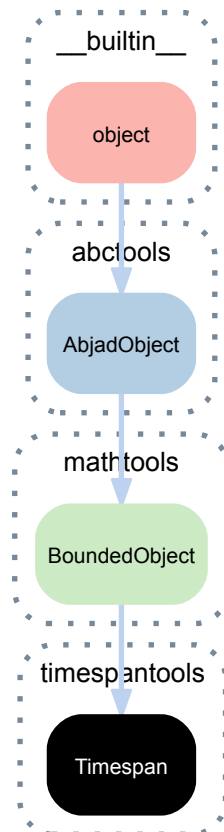
```
>>> print format(simple_inequality)
timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset')
```

Returns string.

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

25.2.4 timespantools.Timespan



class timespantools.Timespan (*start_offset=NegativeInfinity, stop_offset=Infinity*)
 A 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)
    
```

Timespans are closed-open intervals.

Timespans are immutable.

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`

Duration of timespan.

```
>>> 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.

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*)

Divides 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*)

Gets 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.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_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_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*)
 Reflects 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))
```

Returns new timespan.

Timespan.round_offsets (*multiplier, anchor=Left, must_be_well_formed=True*)
 Rounds 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))
```

Returns new timespan.

`Timespan.scale` (*multiplier*, *anchor=Left*)
Scales 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))
```

Returns new timespan.

`Timespan.set_duration` (*duration*)
Sets 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))
```

Returns new timespan.

`Timespan.set_offsets` (*start_offset=None*, *stop_offset=None*)
Sets 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))
```

Subtracts 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))
```

Subtracts 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))
```

Returns new 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
>>> 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)`

Stretches 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)`

Translates timespan by *translation*.

```
>>> timespan = timespantools.Timespan(5, 10)
```

```
>>> timespan.translate(2)
Timespan(start_offset=Offset(7, 1), stop_offset=Offset(12, 1))
```

Returns new timespan.

`Timespan.translate_offsets` (*start_offset_translation=None, stop_offset_translation=None*)
 Translates 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))
```

Returns new 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.__format__` (*format_specification*='')
Formats timespan.

Set *format_specification* to '' or 'storage'.

```
>>> print format(timespan_1)
timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(10, 1),
)
```

Returns string.

`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.__makenew__(*args, **kwargs)`
 Makes new timespan with *args* and *kwargs*.

```
>>> new(timespan_1, stop_offset=Offset(9))
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(9, 1))
```

Returns new timespan.

`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 format(new_timespan)
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(12, 1),
        ),
    ]
)
```

```
>>> new_timespan = timespan_1 | timespan_3
>>> print format(new_timespan)
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(-2, 1),
            stop_offset=durationtools.Offset(10, 1),
        ),
    ]
)
```

```
>>> new_timespan = timespan_1 | timespan_4
>>> print format(new_timespan)
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(20, 1),
        ),
    ]
)
```

```
>>> new_timespan = timespan_2 | timespan_3
>>> print format(new_timespan)
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 format(new_timespan)
timespantools.TimespanInventory(
  [
    timespantools.Timespan(
      start_offset=durationtools.Offset(5, 1),
      stop_offset=durationtools.Offset(20, 1),
    ),
  ]
)
```

```
>>> new_timespan = timespan_3 | timespan_4
>>> print format(new_timespan)
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.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

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 XOR 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 format(new_timespan)
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 format(new_timespan)
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 format(new_timespan)
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 format(new_timespan)
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 format(new_timespan)
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),
        ),
    ]
)

```

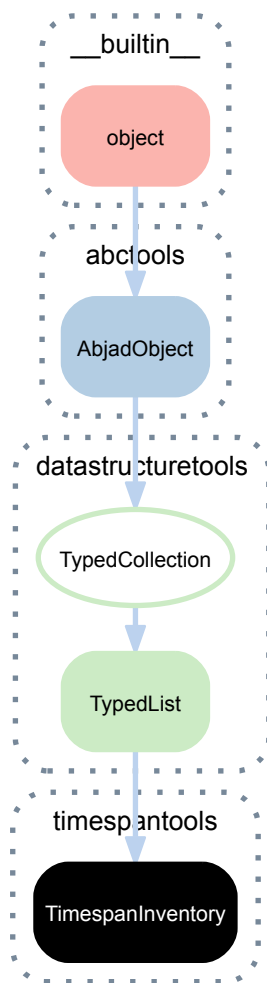
```

>>> new_timespan = timespan_3 ^ timespan_4
>>> print format(new_timespan)
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.

25.2.5 timespantools.TimespanInventory



class `timespantools.TimespanInventory` (*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

A timespan inventory.

Example 1:

```
>>> timespan_inventory_1 = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])
```

```
>>> print format(timespan_inventory_1)
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 format(timespan_inventory_2)
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 format(timespan_inventory_3)
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.

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) .**custom_identifier**
Gets and sets custom identifier of typed collection.

Returns string or none.

(TypedList) .**keep_sorted**
Sorts collection on mutation if true.

Methods

(TypedList) .**append** (*token*)
Changes *token* to item and appends.

```
>>> 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 format(timespan_inventory)  
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 format(timespan_inventory)  
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 format(timespan_inventory)
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 format(timespan_inventory)
timespantools.TimespanInventory(
    []
)
```

Example 2:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     ])
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> print format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
timespantools.TimespanInventory(
    []
)
```

Example 2:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
... ])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> print format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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)`
 Changes *token* to item and returns 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 format(timespan_inventory_1)  
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 format(timespan_inventory_2)  
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)
```



```
        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 format(exploded_inventory)
...
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 format(exploded_inventory)
...
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*)

Changes *tokens* to items and extends.

```

>>> 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 = \
...     timespantools.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 = \
...     timespantools.timespan_2_starts_during_timespan_1(
...         timespan_1=timespan_1)

```

```
>>> result = \
...     timespan_inventory_1.get_timespans_that_satisfy_time_relation(
...     time_relation)

>>> print format(result)
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 = \
...     timespantools.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 = \
...     timespantools.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*)

Changes *token* to item and returns index.

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

Returns index.

(`TypedList`) **.insert** (*i*, *token*)

Changes *token* to item and inserts.

```
>>> 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.

`TimespanInventory.partition(include_tangent_timespans=False)`

Partition timespans into inventories:

Example 1:

```
>>> print format(timespan_inventory_1)
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 format(inventory)
...
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 format(timespan_inventory_2)
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 format(inventory)
...
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 format(inventory)
...
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 format(timespan_inventory)
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 format(timespan_inventory)
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*)

Changes *token* to item and removes.

```
>>> 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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(rounded_inventory)
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 format(rounded_inventory)
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 format(rounded_inventory)
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 format(rounded_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(left)
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 format(right)
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 format(left)
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 format(right)
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 format(left)
timespantools.TimespanInventory(
    []
)
```

```
>>> print format(right)
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 format(inventory)
...
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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 format(timespan_inventory)
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*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

(TypedList).**__delitem__**(*i*)

Aliases list.**__delitem__**().

(TypedCollection).**__eq__**(*expr*)

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

(TypedCollection).**__format__**(*format_specification*='')

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedList).**__getitem__**(*i*)

Aliases list.**__getitem__**().

(TypedList).**__iadd__**(*expr*)

Changes tokens in *expr* to items and extends.

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

```
>>> print format(dynamic_collection)
datastructuretools.TypedList(
  [
    indicatortools.Dynamic(
      'ppp'
    ),
    indicatortools.Dynamic(
      'p'
    ),
    indicatortools.Dynamic(
      'mp'
    ),
    indicatortools.Dynamic(
      'mf'
    ),
    indicatortools.Dynamic(
      'fff'
    ),
  ],
)
```

```

        item_class=indicatortools.Dynamic,
    )

```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

(TypedList).**__makenew__**(*tokens=None, item_class=None, keep_sorted=None, custom_identifier=None*)

Makes new typed list.

Returns new typed list.

(TypedCollection).**__ne__**(*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TypedList).**__reversed__**()

Aliases list.**__reversed__**().

(TypedList).**__setitem__**(*i, expr*)

Changes tokens in *expr* to items and sets.

```

>>> pitch_collection[-1] = 'gqs,'
>>> print format(pitch_collection)
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 format(pitch_collection)
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),

```

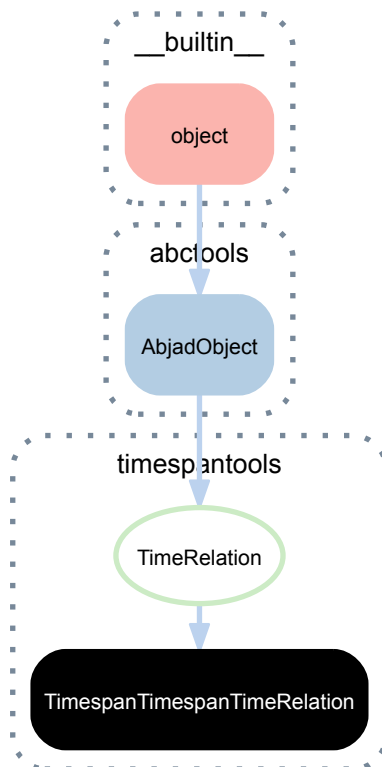
```
...    timespantools.Timespan(-2, 8),
...    ])
```

```
>>> timespan = timespantools.Timespan(5, 10)
>>> result = timespan_inventory - timespan
```

```
>>> print format(timespan_inventory)
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.

25.2.6 timespantools.TimespanTimespanTimeRelation



```
class timespantools.TimespanTimespanTimeRelation (inequality=None, 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:**

```
>>> timespantools.timespan_2_happens_during_timespan_1(
...     timespan_1=last_tuplet,
...     timespan_2=long_note,
...     )
False
```

Example 2:

```
>>> timespantools.timespan_2_intersects_timespan_1(
...     timespan_1=last_tuplet,
...     timespan_2=long_note,
...     )
True
```

Example 3:

```
>>> timespantools.timespan_2_is_congruent_to_timespan_1(
...     timespan_1=last_tuplet,
...     timespan_2=long_note,
...     )
False
```

Example 4:

```
>>> timespantools.timespan_2_overlaps_all_of_timespan_1(
...     timespan_1=last_tuplet,
...     timespan_2=long_note,
...     )
False
```

Example 5:

```
>>> timespantools.timespan_2_overlaps_start_of_timespan_1(
...     timespan_1=last_tuplet,
...     timespan_2=long_note,
...     )
True
```

Example 6:

```
>>> timespantools.timespan_2_overlaps_stop_of_timespan_1(
...     timespan_1=last_tuplet,
...     timespan_2=long_note,
...     )
False
```

Example 7:

```
>>> timespantools.timespan_2_starts_after_timespan_1_starts(
...     timespan_1=last_tuplet,
...     timespan_2=long_note,
...     )
False
```

Example 8:

```
>>> timespantools.timespan_2_starts_after_timespan_1_stops(
...     timespan_1=last_tuplet,
...     timespan_2=long_note,
... )
False
```

Timespan / timespan time relations are immutable.

Bases

- `timespantools.TimeRelation`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(`TimeRelation`).**`inequality`**

Time relation inequality.

Return 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 = \
...     timespantools.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.

`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 = \
...     timespantools.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 = \
...     timespantools.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 = \
...     timespantools.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.

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 = timespantools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1,
...     timespan_2=timespan_2,
...     hold=True,
... )

>>> print format(time_relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
            timespantools.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 = \
...     timespantools.timespan_2_starts_during_timespan_1()
```

```
>>> time_relation_2 = \
...     timespantools.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.

(TimeRelation).**__format__**(*format_specification*='')
Formats time relation.

Returns string.

(TimeRelation).**__makenew__**(*args, **kwargs)
Makes new time relation with optional *args* and *kwargs*.

Returns new time relation.

(AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.

Returns string.

25.3 Functions

25.3.1 timespantools.offset_happens_after_timespan_starts

`timespantools.offset_happens_after_timespan_starts` (*timespan=None*, *offset=None*,
hold=False)

Makes time relation indicating that *offset* happens after *timespan* starts.

```
>>> relation = timespantools.offset_happens_after_timespan_starts()
>>> print format(relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan.start < offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.2 timespantools.offset_happens_after_timespan_stops

`timespantools.offset_happens_after_timespan_stops` (*timespan=None*, *offset=None*,
hold=False)

Makes time relation indicating that *offset* happens after *timespan* stops.

```
>>> relation = timespantools.offset_happens_after_timespan_stops()
>>> print format(relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
```

```
        timespantools.SimpleInequality('timespan.stop < offset'),
    ],
    logical_operator='and',
),
)
```

Returns time relation or boolean.

25.3.3 timespantools.offset_happens_before_timespan_starts

`timespantools.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 = timespantools.offset_happens_before_timespan_starts()
>>> print format(relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.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 = \
...     timespantools.offset_happens_before_timespan_starts(
...         offset=offset)
```

```
>>> print format(relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.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 = \
...     timespantools.offset_happens_before_timespan_starts(
...         timespan=timespan)
```

```
>>> print format(relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.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 = timespantools.offset_happens_before_timespan_starts(
...     timespan=timespan,
...     offset=offset,
...     hold=True,
... )

>>> print format(relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.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:

```
>>> timespantools.offset_happens_before_timespan_starts(
...     timespan=timespan,
...     offset=offset,
...     hold=False,
... )
True
```

Returns time relation or boolean.

25.3.4 timespantools.offset_happens_before_timespan_stops

`timespantools.offset_happens_before_timespan_stops` (*timespan=None, offset=None, hold=False*)

Makes time relation indicating that *offset* happens before *timespan* stops.

```
>>> relation = timespantools.offset_happens_before_timespan_stops()
>>> print format(relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('offset < timespan.stop'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.5 timespantools.offset_happens_during_timespan

`timespantools.offset_happens_during_timespan` (*timespan=None, offset=None, hold=False*)

Makes time relation indicating that *offset* happens during *timespan*.

```
>>> relation = timespantools.offset_happens_during_timespan()
>>> print format(relation)
timespantools.OffsetTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan.start <= offset'),
            timespantools.SimpleInequality('offset < timespan.stop'),
        ],
        logical_operator='and',
    ),
)
```

```
    ),  
    )
```

Returns time relation or boolean.

25.3.6 `timespantools.offset_happens_when_timespan_starts`

`timespantools.offset_happens_when_timespan_starts` (*timespan=None*, *offset=None*,
 hold=False)

Makes time relation indicating that *offset* happens when *timespan* starts.

```
>>> relation = timespantools.offset_happens_when_timespan_starts()  
>>> print format(relation)  
timespantools.OffsetTimespanTimeRelation(  
    inequality=timespantools.CompoundInequality(  
        [  
            timespantools.SimpleInequality('offset == timespan.start'),  
        ],  
        logical_operator='and',  
    ),  
)
```

Returns time relation or boolean.

25.3.7 `timespantools.offset_happens_when_timespan_stops`

`timespantools.offset_happens_when_timespan_stops` (*timespan=None*, *offset=None*,
 hold=False)

Makes time relation indicating that *offset* happens when *timespan* stops.

```
>>> relation = timespantools.offset_happens_when_timespan_stops()  
>>> print format(relation)  
timespantools.OffsetTimespanTimeRelation(  
    inequality=timespantools.CompoundInequality(  
        [  
            timespantools.SimpleInequality('offset == timespan.stop'),  
        ],  
        logical_operator='and',  
    ),  
)
```

Returns time relation or boolean.

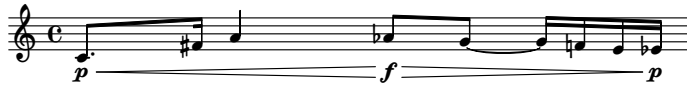
25.3.8 `timespantools.timespan_2_contains_timespan_1_improperly`

`timespantools.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.

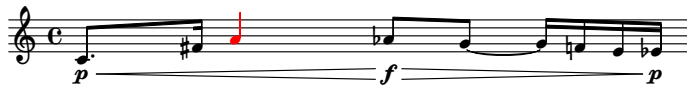
```
>>> relation = timespantools.timespan_2_contains_timespan_1_improperly()  
>>> print format(relation)  
timespantools.TimespanTimespanTimeRelation(  
    inequality=timespantools.CompoundInequality(  
        [  
            timespantools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),  
            timespantools.SimpleInequality('timespan_1.stop_offset <= timespan_2.stop_offset'),  
        ],  
        logical_operator='and',  
    ),  
)
```

```
>>> 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 = timespantools.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.

25.3.9 timespantools.timespan_2_curtails_timespan_1

`timespantools.timespan_2_curtails_timespan_1` (*timespan_1=None, timespan_2=None, hold=False*)

Makes time relation indicating that *timespan_2* curtails *timespan_1*.

```
>>> relation = timespantools.timespan_2_curtails_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset'),
            timespantools.SimpleInequality('timespan_2.start_offset <= timespan_1.stop_offset'),
            timespantools.SimpleInequality('timespan_1.stop_offset <= timespan_2.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.10 timespantools.timespan_2_delays_timespan_1

`timespantools.timespan_2_delays_timespan_1` (*timespan_1=None, timespan_2=None, hold=False*)

Makes time relation indicating that *timespan_2* delays *timespan_1*.

```
>>> relation = timespantools.timespan_2_delays_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
            timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.11 timespantools.timespan_2_happens_during_timespan_1

`timespantools.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 = timespantools.timespan_2_happens_during_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
            timespantools.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))
>>> timespantools.timespan_2_happens_during_timespan_1(
...     timespan_1=timespan_1,
...     timespan_2=timespan_2,
... )
True
```

Returns time relation or boolean.

25.3.12 timespantools.timespan_2_intersects_timespan_1

`timespantools.timespan_2_intersects_timespan_1` (*timespan_1=None*, *timespan_2=None*, *hold=False*)

Makes time relation indicating that *timespan_2* intersects *timespan_1*.

```
>>> relation = timespantools.timespan_2_intersects_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.CompoundInequality(
                [
                    timespantools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
                    timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
                ],
                logical_operator='and',
            ),
            timespantools.CompoundInequality(
                [
                    timespantools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
                    timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
                ],
                logical_operator='and',
            ),
        ],
        logical_operator='or',
    ),
)
```

Returns time relation or boolean.

25.3.13 timespantools.timespan_2_is_congruent_to_timespan_1

`timespantools.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 = timespantools.timespan_2_is_congruent_to_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset == timespan_2.start_offset'),
            timespantools.SimpleInequality('timespan_1.stop_offset == timespan_2.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.14 timespantools.timespan_2_overlaps_all_of_timespan_1

`timespantools.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 = timespantools.timespan_2_overlaps_all_of_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
            timespantools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.15 timespantools.timespan_2_overlaps_only_start_of_timespan_1

`timespantools.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 = timespantools.timespan_2_overlaps_only_start_of_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
            timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
            timespantools.SimpleInequality('timespan_2.stop_offset <= timespan_1.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.16 timespantools.timespan_2_overlaps_only_stop_of_timespan_1

`timespantools.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 = timespantools.timespan_2_overlaps_only_stop_of_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
  inequality=timespantools.CompoundInequality(
    [
      timespantools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
      timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
      timespantools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset'),
    ],
    logical_operator='and',
  ),
)
```

Returns time relation or boolean.

25.3.17 timespantools.timespan_2_overlaps_start_of_timespan_1

`timespantools.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 = timespantools.timespan_2_overlaps_start_of_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
  inequality=timespantools.CompoundInequality(
    [
      timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
      timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
    ],
    logical_operator='and',
  ),
)
```

Returns time relation or boolean.

25.3.18 timespantools.timespan_2_overlaps_stop_of_timespan_1

`timespantools.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 = timespantools.timespan_2_overlaps_stop_of_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
  inequality=timespantools.CompoundInequality(
    [
      timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
      timespantools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset'),
    ],
    logical_operator='and',
  ),
)
```

Returns time relation or boolean.

25.3.19 timespantools.timespan_2_starts_after_timespan_1_starts

`timespantools.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 = timespantools.timespan_2_starts_after_timespan_1_starts()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.20 timespantools.timespan_2_starts_after_timespan_1_stops

`timespantools.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 = timespantools.timespan_2_starts_after_timespan_1_stops()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.stop_offset <= timespan_2.start_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.21 timespantools.timespan_2_starts_before_timespan_1_starts

`timespantools.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 = timespantools.timespan_2_starts_before_timespan_1_starts()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.22 timespantools.timespan_2_starts_before_timespan_1_stops

`timespantools.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 = timespantools.timespan_2_starts_before_timespan_1_stops()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
```

```
[
    timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
],
logical_operator='and',
),
)
```

Returns time relation or boolean.

25.3.23 timespantools.timespan_2_starts_during_timespan_1

`timespantools.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 = timespantools.timespan_2_starts_during_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
            timespantools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

```
>>> 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 = \
...     timespantools.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.

25.3.24 timespantools.timespan_2_starts_when_timespan_1_starts

`timespantools.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 = timespantools.timespan_2_starts_when_timespan_1_starts()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset == timespan_2.start_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.25 timespantools.timespan_2_starts_when_timespan_1_stops

`timespantools.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 = timespantools.timespan_2_starts_when_timespan_1_stops()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_2.start_offset == timespan_1.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.26 timespantools.timespan_2_stops_after_timespan_1_starts

`timespantools.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 = timespantools.timespan_2_stops_after_timespan_1_starts()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.27 `timespantools.timespan_2_stops_after_timespan_1_stops`

`timespantools.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 = timespantools.timespan_2_stops_after_timespan_1_stops()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
  inequality=timespantools.CompoundInequality(
    [
      timespantools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset'),
    ],
    logical_operator='and',
  ),
)
```

Returns time relation or boolean.

25.3.28 `timespantools.timespan_2_stops_before_timespan_1_starts`

`timespantools.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 = timespantools.timespan_2_stops_before_timespan_1_starts()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
  inequality=timespantools.CompoundInequality(
    [
      timespantools.SimpleInequality('timespan_2.stop_offset < timespan_1.start_offset'),
    ],
    logical_operator='and',
  ),
)
```

Returns time relation or boolean.

25.3.29 `timespantools.timespan_2_stops_before_timespan_1_stops`

`timespantools.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 = timespantools.timespan_2_stops_before_timespan_1_stops()
>>> print format(time_relation)
timespantools.TimespanTimespanTimeRelation(
  inequality=timespantools.CompoundInequality(
    [
      timespantools.SimpleInequality('timespan_2.stop_offset < timespan_1.stop_offset'),
    ],
    logical_operator='and',
  ),
)
```

Returns time relation or boolean.

25.3.30 timespantools.timespan_2_stops_during_timespan_1

`timespantools.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 = timespantools.timespan_2_stops_during_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
            timespantools.SimpleInequality('timespan_2.stop_offset <= timespan_1.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.31 timespantools.timespan_2_stops_when_timespan_1_starts

`timespantools.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 = timespantools.timespan_2_stops_when_timespan_1_starts()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.32 timespantools.timespan_2_stops_when_timespan_1_stops

`timespantools.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 = timespantools.timespan_2_stops_when_timespan_1_stops()
>>> print format(inequality)
timespantools.TimespanTimespanTimeRelation(
    inequality=timespantools.CompoundInequality(
        [
            timespantools.SimpleInequality('timespan_2.stop_offset == timespan_1.stop_offset'),
        ],
        logical_operator='and',
    ),
)
```

Returns time relation or boolean.

25.3.33 timespantools.timespan_2_trisects_timespan_1

`timespantools.timespan_2_trisects_timespan_1` (*timespan_1=None, timespan_2=None, hold=False*)

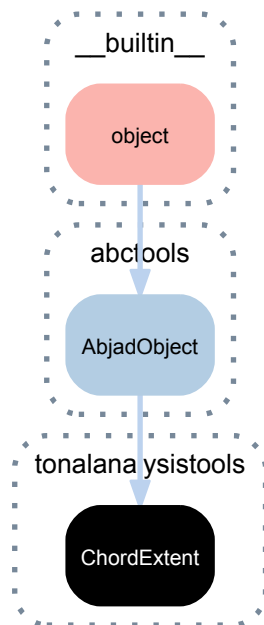
Makes time relation indicating that *timespan_2* trisects *timespan_1*.

```
>>> relation = timespantools.timespan_2_trisects_timespan_1()
>>> print format(relation)
timespantools.TimespanTimespanTimeRelation(
  inequality=timespantools.CompoundInequality(
    [
      timespantools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset'),
      timespantools.SimpleInequality('timespan_2.stop_offset < timespan_1.stop_offset'),
    ],
    logical_operator='and',
  ),
)
```

Returns time relation or boolean.

26.1 Concrete classes

26.1.1 tonalanalysistools.ChordExtent



class `tonalanalysistools.ChordExtent` (*arg*)
A chord extent, 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

`ChordExtent.name`
Name of chord extent.
Returns string.

`ChordExtent.number`
Number of chord extent.

Returns nonnegative integer.

Special methods

`ChordExtent.__eq__(arg)`

True when *arg* is a chord extent with number equal to that of this chord extent. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`ChordExtent.__ne__(arg)`

True when chord extent does not equal *arg*. Otherwise false.

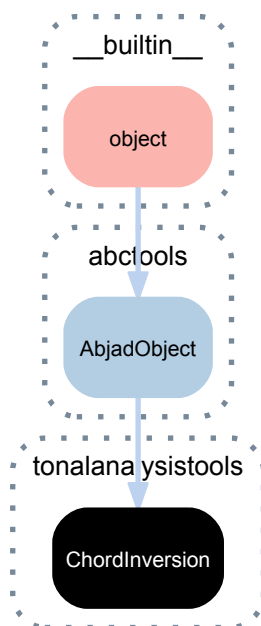
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

26.1.2 tonalanalysistools.ChordInversion



class `tonalanalysistools.ChordInversion` (*arg*=0)

A chord inversion 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

`ChordInversion.name`
Name of chord inversion.

Returns string.

`ChordInversion.number`
Number of chord inversion.

Returns nonnegative integer.

`ChordInversion.title`
Title of chord inversion.

Returns string.

Methods

`ChordInversion.extent_to_figured_bass_string` (*extent*)
Changes *extent* to figured bass string.

Returns string.

Special methods

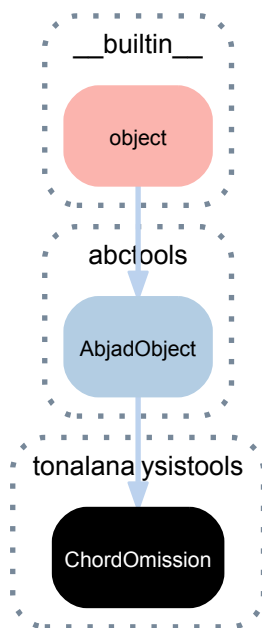
`ChordInversion.__eq__` (*arg*)
True when *arg* is a chord inversion with number equal to that of this chord inversion. Otherwise false.
Returns boolean.

`(AbjadObject).__format__` (*format_specification*=''')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`ChordInversion.__ne__` (*arg*)
True when chord inversion does not equal *arg*. Otherwise false.
Returns boolean.

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

26.1.3 tonalanalysistools.ChordOmission



class `tonalanalysistools.ChordOmission`

A chord omission.

Value object that can not be chnaged after instantiation.

Bases

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

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

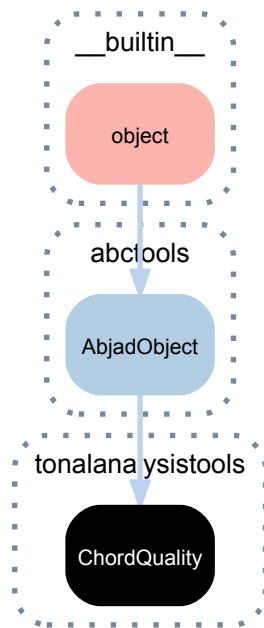
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

26.1.4 tonalanalysisistools.ChordQuality



class `tonalanalysisistools.ChordQuality` (*quality_string*)
 A chord quality, such as major, minor, dominant, diminished and so on.
 Value object that can not be changed after instantiation.

Bases

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

Read-only properties

`ChordQuality.is_uppercase`
 True when chord quality is uppercase. Otherwise false.
 Returns boolean.

`ChordQuality.quality_string`
 Quality string of chord quality.
 Returns string.

Special methods

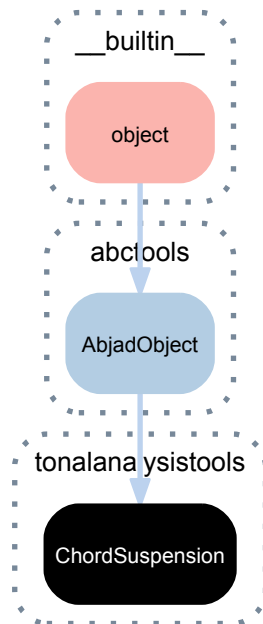
`ChordQuality.__eq__(arg)`
 True when *arg* is a chord quality with quality string equal to that of this chord quality. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`ChordQuality.__ne__(arg)`
 True when chord quality does not equal *arg*. Otherwise false.
 Returns boolean.

`ChordQuality.__repr__()`
 Gets interpreter representation of chord quality.
 Returns string.

26.1.5 tonalanalysistools.ChordSuspension



class `tonalanalysistools.ChordSuspension(*args)`
 A chord of 9-8, 7-6, 4-3, 2-1 and other types of suspension typical of, for example, the Bach chorales.

```
>>> suspension = tonalanalysistools.ChordSuspension(4, 3)
>>> suspension
ChordSuspension(ScaleDegree('4'), ScaleDegree('3'))
```

Value object that can not be changed after instantiation.

Bases

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

Read-only properties

`ChordSuspension.chord_name`
 Chord name of suspension.

```
>>> suspension.chord_name
'sus4'
```

Returns string.

`ChordSuspension.figured_bass_pair`
 Figured bass pair of suspension.

```
>>> suspension.figured_bass_pair
(4, 3)
```

Returns integer pair.

`ChordSuspension.figured_bass_string`
Figured bass string.

```
>>> suspension.figured_bass_string
'4-3'
```

Returns string.

`ChordSuspension.is_empty`
True when start and stop are none. Otherwise false.

```
>>> suspension.is_empty
False
```

`ChordSuspension.start`
Start of suspension.

```
>>> suspension.start
ScaleDegree(4)
```

Returns scale degree.

`ChordSuspension.stop`
Stop of suspension.

```
>>> suspension.stop
ScaleDegree(3)
```

Returns scale degree.

`ChordSuspension.title_string`
Title string of suspension.

```
>>> suspension.title_string
'FourThreeSuspension'
```

Returns string.

Special methods

`ChordSuspension.__eq__(arg)`
True when *arg* is a chord suspension when start and stop equal to those of this chord suspension. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`ChordSuspension.__ne__(arg)`
True when *arg* does not equal chord suspension. Otherwise false.

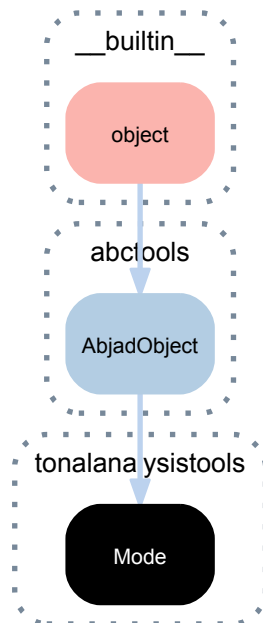
Returns boolean.

`(AbjadObject).__repr__()`
Gets interpreter representation of Abjad object.

Returns string.

`ChordSuspension.__str__()`
 String representation of chord suspension.
 Returns string.

26.1.6 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 name.
 Returns string.

`Mode.named_interval_segment`
 Named interval segment of mode.
 Returns named interval segment.

Special methods

`Mode.__eq__` (*arg*)
 True when *arg* is a mode with mode name equal to that of this mode. Otherwise false.
 Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

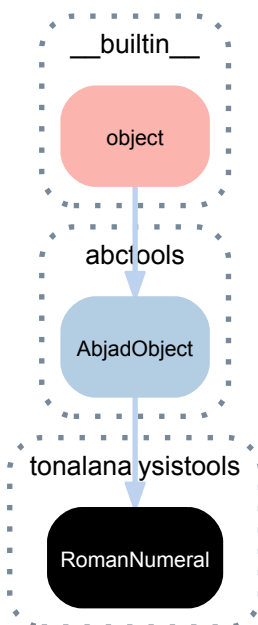
Mode.**__len__**()
 Length of mode.
 Returns nonnegative integer.

Mode.**__ne__**(*arg*)
 True when *arg* does not equal mode. Otherwise false.
 Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

Mode.**__str__**()
 String representation of mode.
 Returns string.

26.1.7 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`

Base scale degree of roman numeral.

Returns scale degree.

`RomanNumeral.extent`

Extend of roman numeral.

Returns extent.

`RomanNumeral.figured_bass_string`

Figured bass string of roman numeral.

Returns string.

`RomanNumeral.inversion`

Inversion of roman numeral.

Returns nonnegative integer.

`RomanNumeral.markup`

Markup of roman numeral.

Returns markup.

`RomanNumeral.quality`

Quality of roman numeral.

Returns chord quality.

`RomanNumeral.root_scale_degree`

Root scale degree.

Returns scale degree.

`RomanNumeral.scale_degree`

Scale degree of roman numeral.

Returns scale degree.

`RomanNumeral.suspension`

Suspension of roman numeral.

Returns suspension.

`RomanNumeral.symbolic_string`

Symbolic string of roman numeral.

Returns string.

Special methods

`RomanNumeral.__eq__(arg)`

True when *arg* is a roman numeral with scale degree, quality, extent, inversion and suspension equal to those of this roman numeral. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`RomanNumeral.__ne__(arg)`

True when roman numeral does not equal *arg*. Otherwise false.

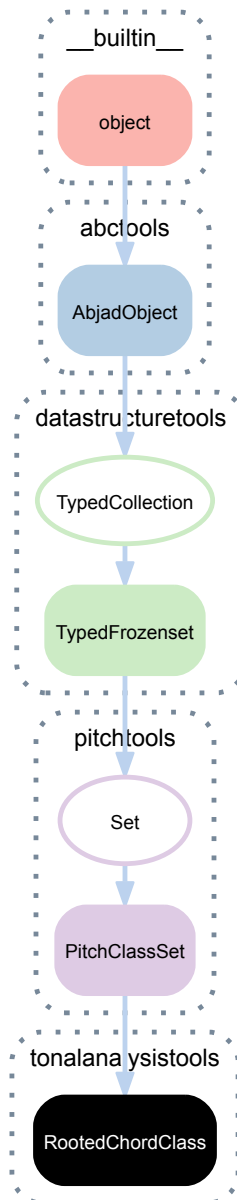
Returns boolean.

`RomanNumeral.__repr__()`

Gets interpreter representation of *arg*.

Returns string.

26.1.8 tonalanalysistools.RootedChordClass



class `tonalanalysistools.RootedChordClass` (*root=None, *args*)

A rooted chord class.

G major triad in root position:

```
>>> tonalanalysistools.RootedChordClass('g', 'major')
GMajorTriadInRootPosition
```

G dominant seventh in root position:

```
>>> chord_class = tonalanalysistools.RootedChordClass('g', 'dominant', 7)
```

Note that notions like a G dominant seventh represent an entire class of chords because there are many different spacings and registrations of a G dominant seventh.

Bases

- `pitchtools.PitchClassSet`
- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`RootedChordClass.bass`
Bass of rooted chord-class.

```
>>> chord_class.bass
NamedPitchClass('g')
```

Returns named pitch-class.

`RootedChordClass.cardinality`
Cardinality of rooted chord-class.

```
>>> chord_class.cardinality
4
```

Returns nonnegative integer.

`RootedChordClass.chord_quality`
Chord quality of rooted chord-class.

```
>>> chord_class.chord_quality
DominantSeventhInRootPosition('P1', '+M3', '+P5', '+m7')
```

Returns chord quality.

`RootedChordClass.extent`
Extent of rooted chord-class.

```
>>> chord_class.extent
ChordExtent(7)
```

Returns chord extent.

`RootedChordClass.figured_bass`
Figured bass of rooted chord-class.

```
>>> chord_class.figured_bass
'7'
```

Returns string.

`RootedChordClass.inversion`
Inversion of rooted chord-class.

```
>>> chord_class.inversion
0
```

Returns nonnegative integer.

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

RootedChordClass.**markup**
Markup of rooted chord-class.

```
>>> show(chord_class.markup)
```

G⁷

Returns markup.

RootedChordClass.**quality_pair**
Quality pair of rooted chord-class.

```
>>> chord_class.quality_pair
('dominant', 'seventh')
```

Returns pair.

RootedChordClass.**root**
Root of rooted chord-class.

```
>>> chord_class.root
NamedPitchClass('g')
```

Returns

RootedChordClass.**root_string**
Root string of rooted chord-class.

```
>>> chord_class.root_string
'G'
```

Returns string.

Read/write properties

(TypedCollection).**custom_identifier**
Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

(TypedFrozenSet).**copy**()
Copies typed frozen set.
Returns new typed frozen set.

(TypedFrozenSet).**difference**(*expr*)
Typed frozen set set-minus *expr*.
Returns new typed frozen set.

(TypedFrozenSet).**intersection**(*expr*)
Set-theoretic intersection of typed frozen set and *expr*.
Returns new typed frozen set.

(PitchClassSet).**invert**()
Inverts 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 pitch-class set is transposed subset of *pcset*. Otherwise false:

```
>>> 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 pitch-class set is transposed superset of *pcset*. Otherwise false:

```
>>> 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)`
True when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenset).issubset(expr)`
True when typed frozen set is a subset of *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenset).issuperset(expr)`
True when typed frozen set is a superset of *expr*. Otherwise false.

Returns boolean.

`(PitchClassSet).multiply(n)`
Multiplies pitch-class set by *n*.

```
>>> pitchtools.PitchClassSet(  
...     [-2, -1.5, 6, 7, -1.5, 7],  
...     ).multiply(5)  
PitchClassSet([2, 4.5, 6, 11])
```

Returns new pitch-class set.

`(PitchClassSet).order_by(pitch_class_segment)`
Orders pitch-class set by *pitch_class_segment*.

Returns pitch-class segment.

`(TypedFrozenset).symmetric_difference(expr)`
Symmetric difference of typed frozen set and *expr*.

Returns new typed frozen set.

`RootedChordClass.transpose()`

Transpose rooted chord-class.

Not yet implemented.

Will return new rooted chord-class.

`(TypedFrozenSet).union(expr)`

Union of typed frozen set and *expr*.

Returns new typed frozen set.

Class methods

`(PitchClassSet).from_selection(selection, item_class=None, custom_identifier=None)`

Makes pitch-class set from *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

`RootedChordClass.cardinality_to_extent(cardinality)`

Change *cardinality* to extent.

Tertian chord with four pitch classes qualifies as a seventh chord:

```
>>> tonalanalysistools.RootedChordClass.cardinality_to_extent(4)
7
```

Returns integer.

`RootedChordClass.extent_to_cardinality(extent)`

Change *extent* to cardinality.

Tertian chord with extent of seven comprises four pitch-classes:

```
>>> tonalanalysistools.RootedChordClass.extent_to_cardinality(7)
4
```

Returns integer.

`RootedChordClass.extent_to_extent_name(extent)`

Change *extent* to extent name.

Extent of seven is a seventh:

```
>>> tonalanalysistools.RootedChordClass.extent_to_extent_name(7)
'seventh'
```

Returns string.

Special methods

`(TypedFrozenSet).__and__(expr)`

Logical AND of typed frozen set and *expr*.

Returns new typed frozen set.

(TypedCollection) .**__contains__** (*token*)

True when typed collection container *token*. Otherwise false.

Returns boolean.

RootedChordClass .**__eq__** (*arg*)

True when *arg* is a rooted chord-class with root, chord quality and inversion equal to those of this rooted chord-class. Otherwise false.

Returns boolean.

(TypedCollection) .**__format__** (*format_specification*='')

Formats typed collection.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TypedFrozenSet) .**__ge__** (*expr*)

True when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

(TypedFrozenSet) .**__gt__** (*expr*)

True when typed frozen set is greater than *expr*. Otherwise false.

Returns boolean.

(PitchClassSet) .**__hash__** ()

Hashes pitch-class set.

Returns integer.

(TypedCollection) .**__iter__** ()

Iterates typed collection.

Returns generator.

(TypedFrozenSet) .**__le__** (*expr*)

True when typed frozen set is less than or equal to *expr*. Otherwise false.

Returns boolean.

(TypedCollection) .**__len__** ()

Length of typed collection.

Returns nonnegative integer.

(TypedFrozenSet) .**__lt__** (*expr*)

True when typed frozen set is less than *expr*. Otherwise false.

Returns boolean.

(TypedCollection) .**__makenew__** (*tokens=None, item_class=None, custom_identifier=None*)

Makes new typed collection with optional new values.

Returns new typed collection.

RootedChordClass .**__ne__** (*arg*)

True when rooted chord-class does not equal *arg*. Otherwise false.

Returns boolean.

(TypedFrozenSet) .**__or__** (*expr*)

Logical OR of typed frozen set and *expr*.

Returns new typed frozen set.

RootedChordClass .**__repr__** ()

Gets interpreter representation of rooted chord-class.

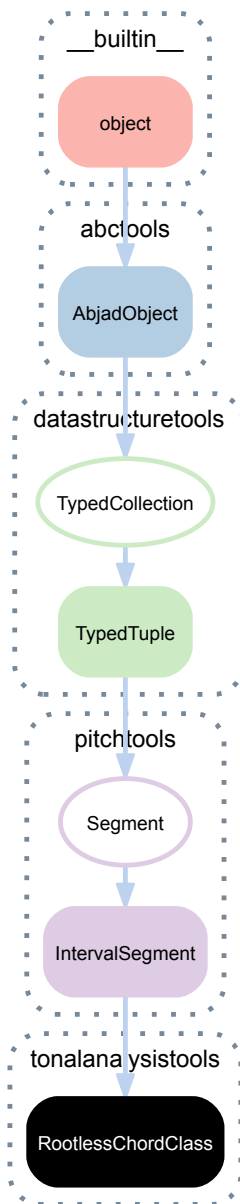
Returns string.

(Set) .**__str__**()
String representation of set.
Returns string.

(TypedFrozenSet) .**__sub__**(*expr*)
Subtracts *expr* from typed frozen set.
Returns new typed frozen set.

(TypedFrozenSet) .**__xor__**(*expr*)
Logical XOR of typed frozen set and *expr*.
Returns new typed frozen set.

26.1.9 tonalanalysistools.RootlessChordClass



class tonalanalysistools.**RootlessChordClass** (*quality_string*='major', *extent*='triad', *inversion*='root')

A rootless chord class.

Major triad in root position:

```
>>> tonalanalysistools.RootlessChordClass('major')
MajorTriadInRootPosition('P1', '+M3', '+P5')
```

Dominant seventh in root position:

```
>>> tonalanalysistools.RootlessChordClass('dominant', 7)
DominantSeventhInRootPosition('P1', '+M3', '+P5', '+m7')
```

German augmented sixth in root position:

```
>>> tonalanalysistools.RootlessChordClass('German', 'augmented sixth')
GermanAugmentedSixthInRootPosition('P1', '+M3', '+m3', '+aug2')
```

Bases

- `pitchtools.IntervalSegment`
- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`RootlessChordClass`.**cardinality**

Cardinality of rootless chord-class.

Returns nonnegative integer.

`RootlessChordClass`.**extent**

Extent of rootless chord-class.

Returns nonnegative integer.

`RootlessChordClass`.**extent_name**

Extent name of rootless chord class.

`(IntervalSegment)`.**has_duplicates**

True if segment has duplicate items. Otherwise false.

```
>>> 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.

`RootlessChordClass`.**inversion**

Inversion of rootless chord-class.

Returns nonnegative integer.

`(TypedCollection)`.**item_class**

Item class to coerce tokens into.

`RootlessChordClass.position`

Position of rootless chord-class.

Returns string.

`RootlessChordClass.quality_string`

Quality string of rootless chord class.

Returns string.

`RootlessChordClass.rotation`

Rotation of rootless chord-class.

Returns nonnegative integer.

`(IntervalSegment).slope`

Slope of interval segment.

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`

Spread of interval segment.

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.

Read/write properties

`(TypedCollection).custom_identifier`

Gets and sets custom identifier of typed collection.

Returns string or none.

Methods

`(TypedTuple).count(token)`

Changes *token* to item.

Returns count in collection.

`(TypedTuple).index(token)`

Changes *token* to item.

Returns index in collection.

`(IntervalSegment).rotate(n)`

Rotates interval segment by *n*.

Returns new interval segment.

Class methods

`(IntervalSegment).from_selection(selection, item_class=None, custom_identifier=None)`
Makes 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

`RootlessChordClass.from_interval_class_segment(segment)`
Makes new rootless chord-class from *segment*.

Returns new rootless chord-class.

Special methods

`(TypedTuple).__add__(expr)`
Adds typed tuple to *expr*.

Returns new typed tuple.

`(TypedTuple).__contains__(token)`
Change *token* to item and return true if item exists in collection.

Returns none.

`(TypedCollection).__eq__(expr)`
True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`
Formats typed collection.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(TypedTuple).__getitem__(i)`
Gets *i* from type tuple.

Returns item.

`(TypedTuple).__getslice__(start, stop)`
Gets slice from *start* to *stop* in typed tuple.

Returns new typed tuple.

`(TypedTuple).__hash__()`
Hashes typed tuple.

Returns integer.

`(TypedCollection).__iter__()`
Iterates typed collection.

Returns generator.

`(TypedCollection).__len__()`
Length of typed collection.

Returns nonnegative integer.

(TypedCollection) .**__makenew__** (*tokens=None, item_class=None, custom_identifier=None*)

Makes new typed collection with optional new values.

Returns new typed collection.

(TypedTuple) .**__mul__** (*expr*)

Multiplies typed tuple by *expr*.

Returns new typed tuple.

(TypedCollection) .**__ne__** (*expr*)

True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.

Returns boolean.

RootlessChordClass .**__repr__** ()

Gets interpreter representation of rootless chord-class.

Returns string.

(TypedTuple) .**__rmul__** (*expr*)

Multiplies *expr* by typed tuple.

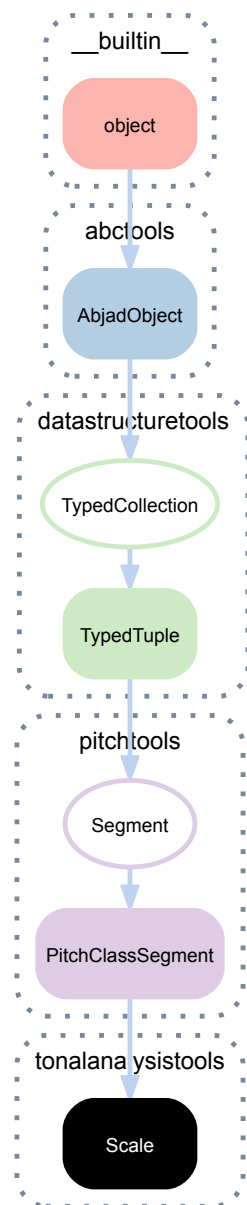
Returns new typed tuple.

(Segment) .**__str__** ()

String representation of segment.

Returns string.

26.1.10 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`

Dominant of scale.

Returns pitch-class.

`(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`

Key signature of scale.

Returns key signature.

`Scale.leading_tone`

Leading tone of scale.

Returns pitch-class.

`Scale.mediant`

Mediant of scale.

Returns pitch-class.

`Scale.named_interval_class_segment`

Named interval class segment of scale.

Returns interval-class segment.

`Scale.subdominant`

Subdominant of scale.

Returns pitch-class.

`Scale.submediant`

Submediate of scale.

Returns pitch-class.

`Scale.superdominant`

Superdominant of scale.

Returns pitch-class.

`Scale.tonic`

Tonic of scale.

Returns pitch-class.

Read/write properties

(TypedCollection).**custom_identifier**
Gets and sets custom identifier of typed collection.
Returns string or none.

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])
```

Returns new pitch-class segment.

(TypedTuple).**count**(*token*)
Changes *token* to item.
Returns count in collection.

Scale.**create_named_pitch_set_in_pitch_range**(*pitch_range*)
Creates named pitch-set in *pitch_range*.
Returns pitch-set.

(TypedTuple).**index**(*token*)
Changes *token* to item.
Returns 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])
```

Returns 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)
Makes first *n* notes in ascending diatonic scale according to *key_signature*.
Set *written_duration* equal to *written_duration* or 1/8:

```
>>> scale = tonalanalysisistools.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 = TimeSignature((5, 4))
>>> attach(time_signature, 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])
```

Returns new pitch-class segment.

Scale **.named_pitch_class_to_scale_degree(*args)**

Changes named pitch-class to scale degree.

Returns scale degree.

(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])
```

Returns 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'])
```

Returns new pitch-class segment.

`Scale.scale_degree_to_named_pitch_class(*args)`

Changes scale degree to named pitch-class.

Returns named pitch-class.

`(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])
```

Returns new pitch-class segment.

Class methods

`Scale.from_selection(selection, item_class=None, name=None)`

Make scale from *selection*.

Returns new scale.

Special methods

`(TypedTuple).__add__(expr)`

Adds typed tuple to *expr*.

Returns new typed tuple.

`(TypedTuple).__contains__(token)`

Change *token* to item and return true if item exists in collection.

Returns none.

`(TypedCollection).__eq__(expr)`

True when *expr* is a typed collection with items that compare equal to those of this typed collection. Otherwise false.

Returns boolean.

`(TypedCollection).__format__(format_specification='')`

Formats typed collection.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(TypedTuple).__getitem__(i)`

Gets *i* from type tuple.

Returns item.

`(TypedTuple).__getslice__(start, stop)`

Gets slice from *start* to *stop* in typed tuple.

Returns new typed tuple.

`(TypedTuple).__hash__()`

Hashes typed tuple.

Returns integer.

`(TypedCollection).__iter__()`

Iterates typed collection.

Returns generator.

(TypedCollection). **__len__**()
Length of typed collection.
Returns nonnegative integer.

(TypedCollection). **__makenew__**(*tokens=None, item_class=None, custom_identifier=None*)
Makes new typed collection with optional new values.
Returns new typed collection.

(TypedTuple). **__mul__**(*expr*)
Multiplies typed tuple by *expr*.
Returns new typed tuple.

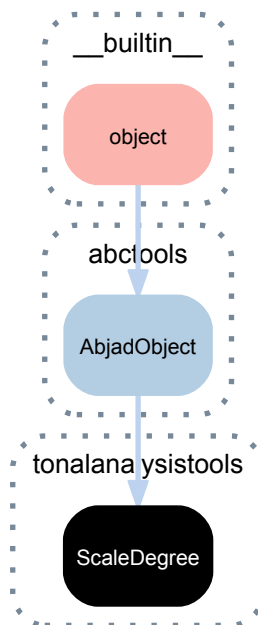
(TypedCollection). **__ne__**(*expr*)
True when *expr* is not a typed collection with items equal to this typed collection. Otherwise false.
Returns boolean.

Scale. **__repr__**()
Gets interpreter representation of scale.
Returns string.

(TypedTuple). **__rmul__**(*expr*)
Multiplies *expr* by typed tuple.
Returns new typed tuple.

(Segment). **__str__**()
String representation of segment.
Returns string.

26.1.11 tonalanalysisistools.ScaleDegree



class tonalanalysisistools.**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.

```

>>> scale_degree = tonalanalysisistools.ScaleDegree('#4')
>>> scale_degree
ScaleDegree('sharp', 4)
  
```

Bases

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

Read-only properties

`ScaleDegree.accidental`
Accidental of scale degree.

```
>>> scale_degree.accidental
Accidental('s')
```

Returns accidental.

`ScaleDegree.name`
Name of scale degree.

```
>>> tonalanalysistools.ScaleDegree(4).name
'subdominant'
```

Returns string.

`ScaleDegree.number`
Number of scale degree.

```
>>> scale_degree.number
4
```

Returns integer from 1 to 7, inclusive.

`ScaleDegree.roman_numeral_string`
Roman numeral string of scale degree.

```
>>> scale_degree.roman_numeral_string
'IV'
```

Returns string.

`ScaleDegree.symbolic_string`
Symbolic string of scale degree.

```
>>> scale_degree.symbolic_string
'#IV'
```

Returns string.

`ScaleDegree.title_string`
Title string of scale degree.

```
>>> scale_degree.title_string
'SharpFour'
```

Returns string.

Methods

`ScaleDegree.apply_accidental(accidental)`
Applies accidental to scale degree.

```
>>> scale_degree.apply_accidental('ff')
ScaleDegree('flat', 4)
```

Returns new scale degree.

Special methods

`ScaleDegree.__eq__(arg)`

True when *arg* is a scale degree with number and accidental equal to those of this scale degree.

```
>>> scale_degree == tonalanalysistools.ScaleDegree('#4')
True
```

Otherwise false:

```
>>> scale_degree == tonalanalysistools.ScaleDegree(4)
False
```

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`ScaleDegree.__ne__(arg)`

True when *arg* does not equal scale degree. Otherwise false.

Returns boolean.

`ScaleDegree.__repr__()`

Gets interpreter representation of scale degree.

```
>>> scale_degree
ScaleDegree('sharp', 4)
```

Returns string.

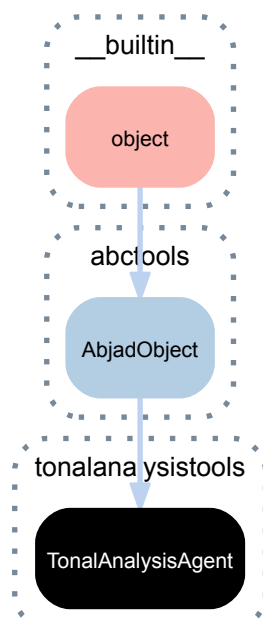
`ScaleDegree.__str__()`

String representation of scale degree.

```
>>> str(scale_degree)
'#4'
```

Returns string.

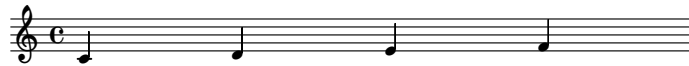
26.1.12 tonalanalysistools.TonalAnalysisAgent



class `tonalanalysistools.TonalAnalysisAgent` (*client=None*)
A tonal analysis interface.

Example 1. Interface to conjunct selection:

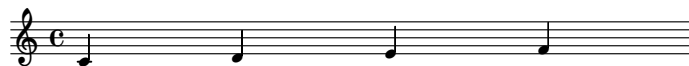
```
>>> staff = Staff("c'4 d' e' f'")
>>> show(staff)
```



```
>>> 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

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

Read-only properties

`TonalAnalysisAgent.client`
Returns client of mutation agent.

Returns selection or component.

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 = KeySignature('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 = KeySignature('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

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`
Gets interpreter representation of Abjad object.

Returns string.

26.2 Functions

26.2.1 tonalanalysistools.select

`tonalanalysistools.select(expr)`
Select *expr* for tonal analysis.

Returns tonal analysis selection.

27.1 Functions

27.1.1 `topleveltools.attach`

`topleveltools.attach` (*indicator*, *component_expression*, *scope=None*)
Attaches *indicator* to *component_expression*.

Derives scope from the default scope of *indicator* when *scope* is none.

Returns none.

27.1.2 `topleveltools.contextualize`

`topleveltools.contextualize` (*expr*)
Applies LilyPond context setting to *expr*.

Returns LilyPond context setting manager.

27.1.3 `topleveltools.detach`

`topleveltools.detach` (*prototype*, *component_expression*)
Detaches from *component_expression* all items matching *prototype*.

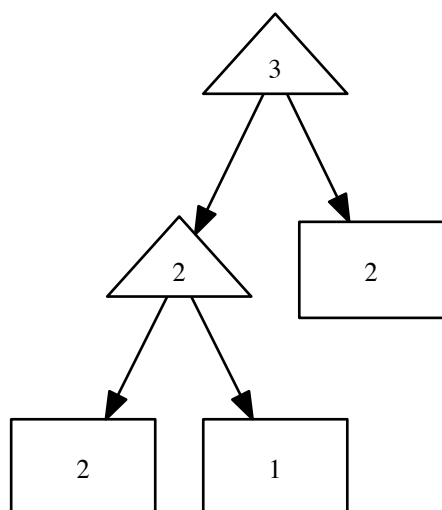
Returns tuple of zero or more detached items.

27.1.4 `topleveltools.graph`

`topleveltools.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))
```

```
>>> topleveltools.graph(rhythm_tree)
```

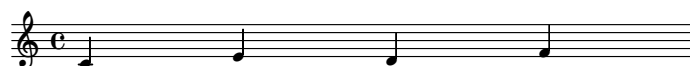


Returns none.

27.1.5 topleveltools.iterate

`topleveltools.iterate(expr)`
iterates *expr*.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



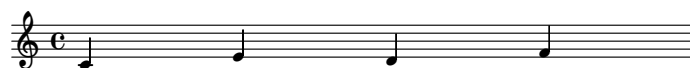
```
>>> notes = staff[-2:]
>>> iterate(notes)
IterationAgent(client=SliceSelection(Note("d'4"), Note("f'4")))
```

Returns score iteration agent.

27.1.6 topleveltools.mutate

`topleveltools.mutate(expr)`
Mutates *expr*.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



```
>>> notes = staff[-2:]
>>> mutate(notes)
MutationAgent(client=SliceSelection(Note("d'4"), Note("f'4")))
```

Returns score mutation agent.

27.1.7 topleveltools.new

`topleveltools.new(expr, *args, **kwargs)`
Makes new *expr* with optionally new *args* and *kwargs*.

Returns new object with the same type as *expr*.

27.1.8 topleveltools.override

`topleveltools.override` (*expr*)
 Overrides *expr*.
 Returns LilyPond grob manager.

27.1.9 topleveltools.parse

`topleveltools.parse` (*arg*, *language*='english')
 Parses *arg* as LilyPond string.

```
>>> parse("{c'4 d'4 e'4 f'4}")
{c'4, d'4, e'4, f'4}
```

```
>>> container = _
```

```
>>> print format(container)
{
    c'4
    d'4
    e'4
    f'4
}
```

A pitch-name language may also be specified.

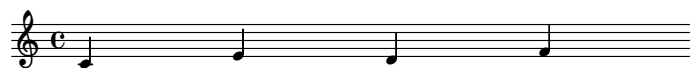
```
>>> parse("{c'8 des' e' fis'}", language='nederlands')
{c'8, df'8, e'8, fs'8}
```

Returns Abjad expression.

27.1.10 topleveltools.persist

`topleveltools.persist` (*expr*)
 Persists *expr*.

```
>>> staff = Staff("c'4 e'4 d'4 f'4")
>>> show(staff)
```



```
>>> persist(staff)
PersistenceAgent(client={c'4, e'4, d'4, f'4})
```

Returns score mutation agent.

27.1.11 topleveltools.play

`topleveltools.play` (*expr*)
 Plays *expr*.

```
>>> note = Note("c'4")
```

```
>>> topleveltools.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.

27.1.12 topleveltools.select

`topleveltools.select` (*expr=None, contiguous=False*)

Selects *expr*.

Returns selection.

27.1.13 topleveltools.show

`topleveltools.show` (*expr, return_timing=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(staff, return_timing=True)
(0, 1)
```



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.

Part II

Demos and example packages

28.1 Functions

28.1.1 `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.

28.1.2 `desordre.make_desordre_lilypond_file`

`desordre.make_desordre_lilypond_file` ()

Makes Désordre LilyPond file.

28.1.3 `desordre.make_desordre_measure`

`desordre.make_desordre_measure` (*pitches*)

Makes 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.

28.1.4 `desordre.make_desordre_pitches`

`desordre.make_desordre_pitches` ()

Makes Désordre pitches.

28.1.5 `desordre.make_desordre_score`

`desordre.make_desordre_score` (*pitches*)

Returns a complete piano staff with Ligeti music.

28.1.6 `desordre.make_desordre_staff`

`desordre.make_desordre_staff` (*pitches*)

Makes Désordre staff.

29.1 Functions

29.1.1 `ferneyhough.configure_lilypond_file`

`ferneyhough.configure_lilypond_file` (*lilypond_file*)
Configures LilyPond file.

29.1.2 `ferneyhough.configure_score`

`ferneyhough.configure_score` (*score*)
Configured score.

29.1.3 `ferneyhough.make_lilypond_file`

`ferneyhough.make_lilypond_file` (*tuple_duration*, *row_count*, *column_count*)
Makes LilyPond file.

29.1.4 `ferneyhough.make_nested_tuplet`

`ferneyhough.make_nested_tuplet` (*tuple_duration*, *outer_tuplet_proportions*, *inner_tuplet_subdivision_count*)
Makes nested tuplet.

29.1.5 `ferneyhough.make_row_of_nested_tuplets`

`ferneyhough.make_row_of_nested_tuplets` (*tuple_duration*, *outer_tuplet_proportions*, *column_count*)
Makes row of nested tuplets.

29.1.6 `ferneyhough.make_rows_of_nested_tuplets`

`ferneyhough.make_rows_of_nested_tuplets` (*tuple_duration*, *row_count*, *column_count*)
Makes rows of nested tuplets.

29.1.7 `ferneyhough.make_score`

`ferneyhough.make_score` (*tuple_duration*, *row_count*, *column_count*)
Makes score.

30.1 Functions

30.1.1 `mozart.choose_mozart_measures`

`mozart.choose_mozart_measures()`
Chooses Mozart measures.

30.1.2 `mozart.make_mozart_lilypond_file`

`mozart.make_mozart_lilypond_file()`
Makes Mozart LilyPond file.

30.1.3 `mozart.make_mozart_measure`

`mozart.make_mozart_measure(measure_dict)`
Makes Mozart measure.

30.1.4 `mozart.make_mozart_measure_corpus`

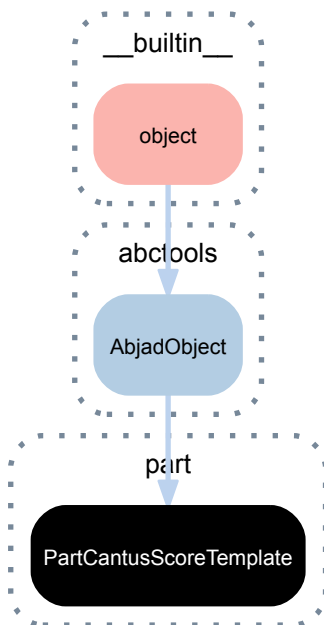
`mozart.make_mozart_measure_corpus()`
Makes Mozart measure corpus.

30.1.5 `mozart.make_mozart_score`

`mozart.make_mozart_score()`
Makes Mozart score.

31.1 Concrete classes

31.1.1 `part.PartCantusScoreTemplate`



`class part.PartCantusScoreTemplate`
Pärt Cantus score template.

Bases

- `abjad.tools.abctools.AbjadObject`
- `__builtin__.object`

Special methods

`PartCantusScoreTemplate.__call__()`

Calls score template.

Returns LilyPond file.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__** (*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__** (*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

31.2 Functions

31.2.1 part.add_bell_music_to_score

`part.add_bell_music_to_score (score)`

Adds bell music to score.

31.2.2 part.add_string_music_to_score

`part.add_string_music_to_score (score)`

Adds string music to score.

31.2.3 part.apply_bowing_marks

`part.apply_bowing_marks (score)`

Applies bowing marks to score.

31.2.4 part.apply_dynamics

`part.apply_dynamics (score)`

Applies dynamics to score.

31.2.5 part.apply_expressive_marks

`part.apply_expressive_marks (score)`

Applies expressive marks to score.

31.2.6 part.apply_final_bar_lines

`part.apply_final_bar_lines (score)`

Applies final bar lines to score.

31.2.7 part.apply_page_breaks

`part.apply_page_breaks (score)`

Applies page breaks to score.

31.2.8 `part.apply_rehearsal_marks`

`part.apply_rehearsal_marks` (*score*)
Applies rehearsal marks to score.

31.2.9 `part.configure_lilypond_file`

`part.configure_lilypond_file` (*lilypond_file*)
Configures LilyPond file.

31.2.10 `part.configure_score`

`part.configure_score` (*score*)
Configures score.

31.2.11 `part.create_pitch_contour_reservoir`

`part.create_pitch_contour_reservoir` ()
Creates pitch contour reservoir.

31.2.12 `part.durate_pitch_contour_reservoir`

`part.durate_pitch_contour_reservoir` (*pitch_contour_reservoir*)
Durates pitch contour reservoir.

31.2.13 `part.edit_bass_voice`

`part.edit_bass_voice` (*score*, *durated_reservoir*)
Edits bass voice.

31.2.14 `part.edit_cello_voice`

`part.edit_cello_voice` (*score*, *durated_reservoir*)
Edits cello voice.

31.2.15 `part.edit_first_violin_voice`

`part.edit_first_violin_voice` (*score*, *durated_reservoir*)
Edits first violin voice.

31.2.16 `part.edit_second_violin_voice`

`part.edit_second_violin_voice` (*score*, *durated_reservoir*)
Edits second violin voice.

31.2.17 `part.edit_viola_voice`

`part.edit_viola_voice` (*score*, *durated_reservoir*)
Edits viola voice.

31.2.18 `part.make_part_lilypond_file`

`part.make_part_lilypond_file()`

Makes Part LilyPond file.

31.2.19 `part.shadow_pitch_contour_reservoir`

`part.shadow_pitch_contour_reservoir(pitch_contour_reservoir)`

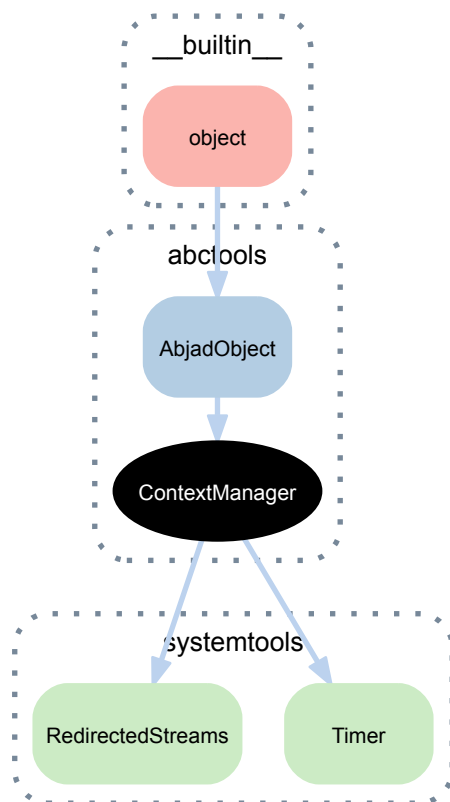
Shadows pitch contour reservoir.

Part III

Abjad internal packages

32.1 Abstract classes

32.1.1 abctools.ContextManager



class `abctools.ContextManager`
An abstract context manager class.

Bases

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

Special methods

`ContextManager.__enter__()`
Enters context manager.

(AbjadObject).**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

ContextManager.**__exit__**(*exc_type, exc_value, traceback*)

Exits context manager.

(AbjadObject).**__format__**(*format_specification=''*)

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

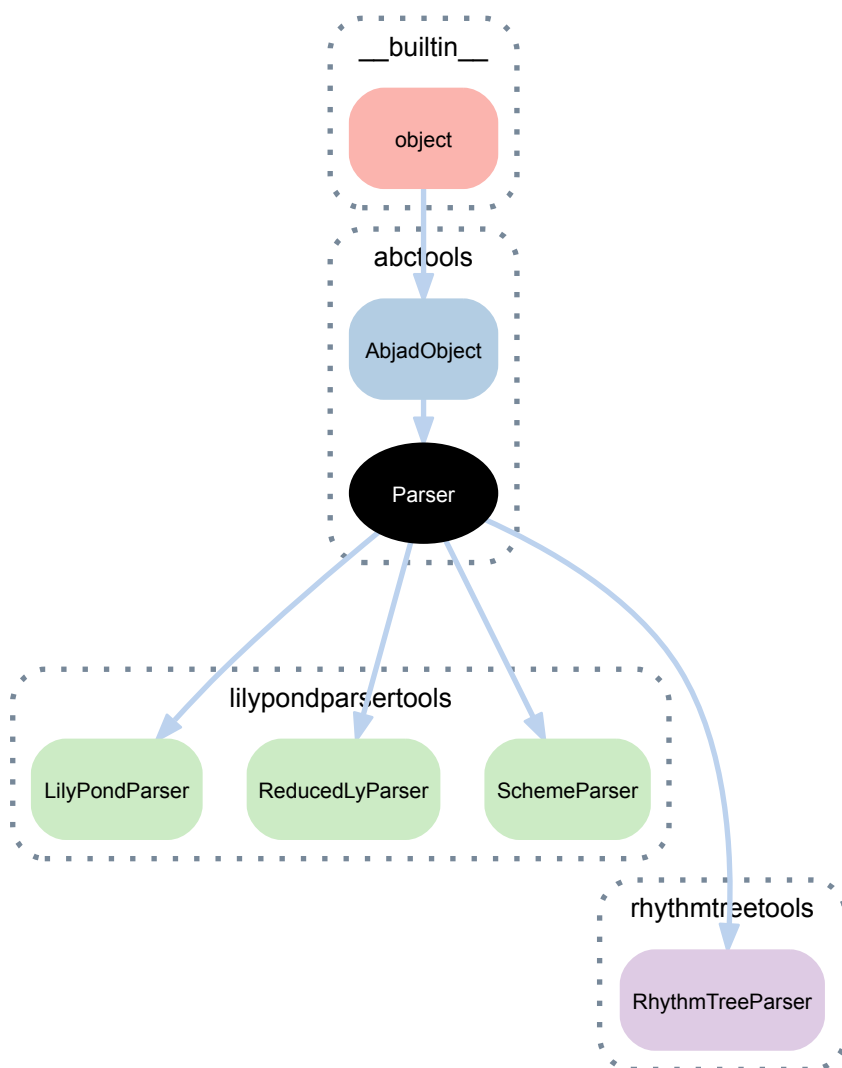
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

32.1.2 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(`AbjadObject`) `.__format__` (*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

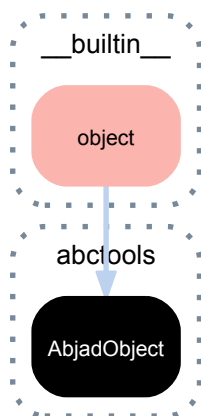
(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

32.2 Concrete classes

32.2.1 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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

AbjadObject.**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

AbjadObject.**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

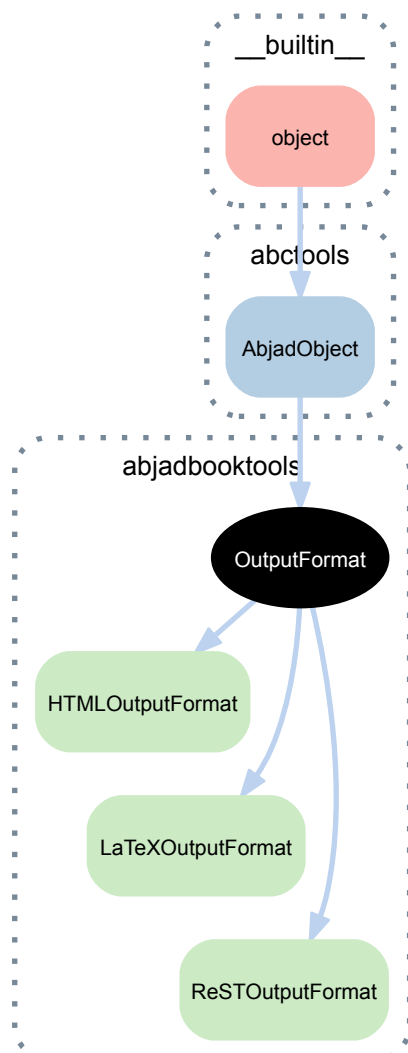
AbjadObject.**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.1 Abstract classes

33.1.1 abjadbooktools.OutputFormat



```
class abjadbooktools.OutputFormat (code_block_opening, code_block_closing, code_indent,
                                   image_block, image_format)
    Output format.
```

Bases

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

Read-only properties

`OutputFormat.code_block_closing`
Code block closing.

`OutputFormat.code_block_opening`
Code block opening.

`OutputFormat.code_indent`
Code indent.

`OutputFormat.image_block`
Image block.

`OutputFormat.image_format`
Image format.

Special methods

`OutputFormat.__call__(code_block, image_dict)`
Calls output format with *code_block* and *image_dict*.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

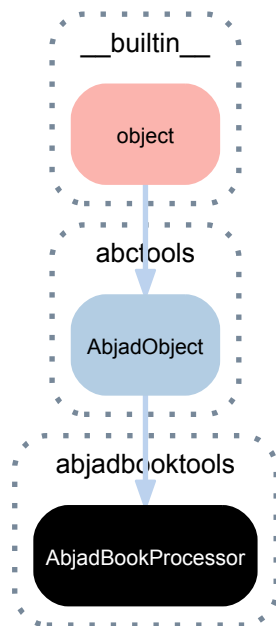
`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to `''` or `'storage'`. Interprets `''` equal to `'storage'`.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

33.2 Concrete classes

33.2.1 abjadbooktools.AbjadBookProcessor



class `abjadbooktools.AbjadBookProcessor` (*directory=None, lines=None, output_format=None, skip_rendering=False, image_prefix='image', verbose=False*)

Abjad book processor.

Bases

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

Read-only properties

`AbjadBookProcessor.directory`
Directory.

`AbjadBookProcessor.image_prefix`
Image prefix.

`AbjadBookProcessor.lines`
Lines.

`AbjadBookProcessor.output_format`
Output format.

`AbjadBookProcessor.skip_rendering`
Skip rendering.

`AbjadBookProcessor.verbose`
Verbose.

Methods

`AbjadBookProcessor.update_status(line)`
Updates status.
Returns none.

Special methods

`AbjadBookProcessor.__call__(verbose=True)`
Calls Abjad book processor.

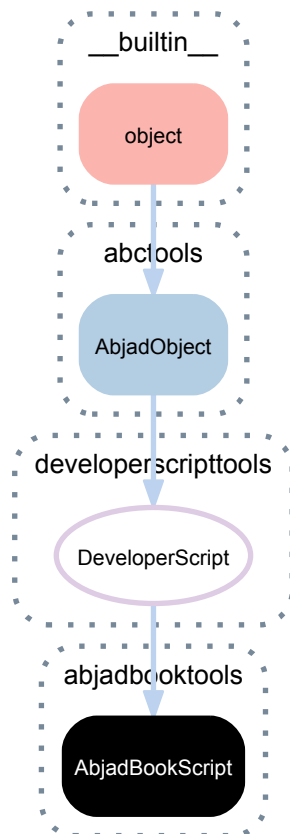
`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

33.2.2 abjadbooktools.AbjadBookScript



class `abjadbooktools.AbjadBookScript`
 abjad book script.

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`AbjadBookScript.alias`
 Alias of Abjad book script.
 Returns 'book'.

`(DeveloperScript).argument_parser`
 The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`
 Colors.
 Returns dictionary.

`(DeveloperScript).formatted_help`
 Formatted help of developer script.

`(DeveloperScript).formatted_usage`
 Formatted usage of developer script.

`(DeveloperScript).formatted_version`
Formatted version of developer script.

`AbjadBookScript.long_description`
Long description of Abjad book script.
Returns string.

`AbjadBookScript.output_formats`
Output formats of Abjad book script.
Returns dictionary.

`(DeveloperScript).program_name`
The name of the script, callable from the command line.

`(DeveloperScript).scripting_group`
Scripting subcommand group of script.

`AbjadBookScript.short_description`
Short description of Abjad book script.
Returns string.

`AbjadBookScript.version`
Version of Abjad book script.
Returns float.

Methods

`AbjadBookScript.process_args (args)`
Processes *args*.
Returns none.

`AbjadBookScript.setup_argument_parser (parser)`
Sets up argument *parser*.
Returns none.

Special methods

`(DeveloperScript).__call__ (args=None)`
Calls developer script.
Returns none.

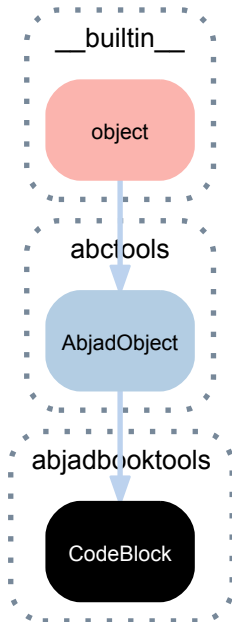
`(AbjadObject).__eq__ (expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__ (format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__ (expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

33.2.3 abjadbooktools.CodeBlock



class `abjadbooktools.CodeBlock` (*lines=None, starting_line_number=0, ending_line_number=1, hide=False, strip_prompt=False*)
 A code block.

Bases

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

Read-only properties

`CodeBlock.ending_line_number`
 Ending line number of code block.

`CodeBlock.hide`
 True when code block should hide.

`CodeBlock.lines`
 Lines of code block.

`CodeBlock.processed_results`
 Processed results of code block.

`CodeBlock.starting_line_number`
 Starting line number of code block.

`CodeBlock.strip_prompt`
 True when code block should strip prompt.

Methods

`CodeBlock.read(pipe)`
Reads *pipe*.

Special methods

`CodeBlock.__call__(processor, pipe, image_count=0, directory=None, image_prefix='image', verbose=False)`
Calls code block.
Returns image count.

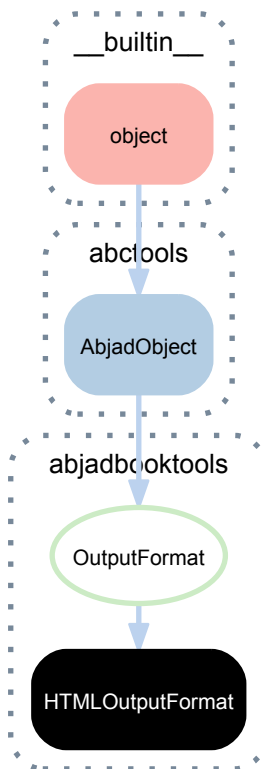
`CodeBlock.__eq__(expr)`
True when *expr* is a code block with lines, starting line number, ending line number, hide and strip prompt boolean equal to those of this code block. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

33.2.4 abjadbooktools.HTMLOutputFormat



class `abjadbooktools.HTMLOutputFormat`
HTML output format.

Bases

- `abjadbooktools.OutputFormat`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(OutputFormat).code_block_closing`
Code block closing.

`(OutputFormat).code_block_opening`
Code block opening.

`(OutputFormat).code_indent`
Code indent.

`(OutputFormat).image_block`
Image block.

`(OutputFormat).image_format`
Image format.

Special methods

`(OutputFormat).__call__(code_block, image_dict)`
Calls output format with *code_block* and *image_dict*.

(AbjadObject) .**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

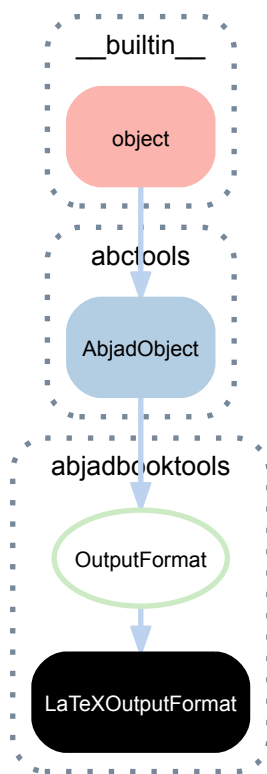
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.2.5 abjadbooktools.LaTeXOutputFormat



class abjadbooktools.LaTeXOutputFormat
 LaTeX output format.

Bases

- abjadbooktools.OutputFormat
- abctools.AbjadObject
- __builtin__.object

Read-only properties

(OutputFormat).**code_block_closing**
Code block closing.

(OutputFormat).**code_block_opening**
Code block opening.

(OutputFormat).**code_indent**
Code indent.

(OutputFormat).**image_block**
Image block.

(OutputFormat).**image_format**
Image format.

Special methods

(OutputFormat).**__call__**(*code_block*, *image_dict*)
Calls output format with *code_block* and *image_dict*.

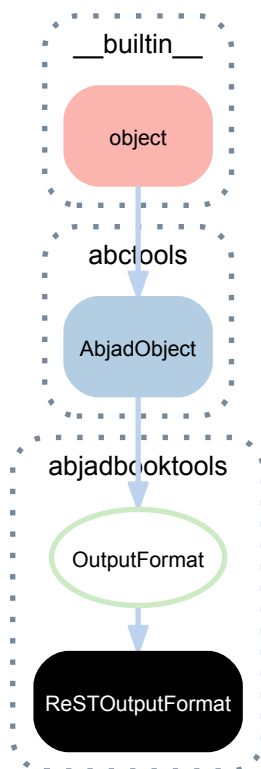
(AbjadObject).**__eq__**(*expr*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

33.2.6 abjadbooktools.ReSTOutputFormat



class `abjadbooktools.ReSTOutputFormat`
ReST output format.

Bases

- `abjadbooktools.OutputFormat`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(OutputFormat).code_block_closing`
Code block closing.

`(OutputFormat).code_block_opening`
Code block opening.

`(OutputFormat).code_indent`
Code indent.

`(OutputFormat).image_block`
Image block.

`(OutputFormat).image_format`
Image format.

Special methods

`(OutputFormat).__call__(code_block, image_dict)`
Calls output format with *code_block* and *image_dict*.

(AbjadObject) .**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

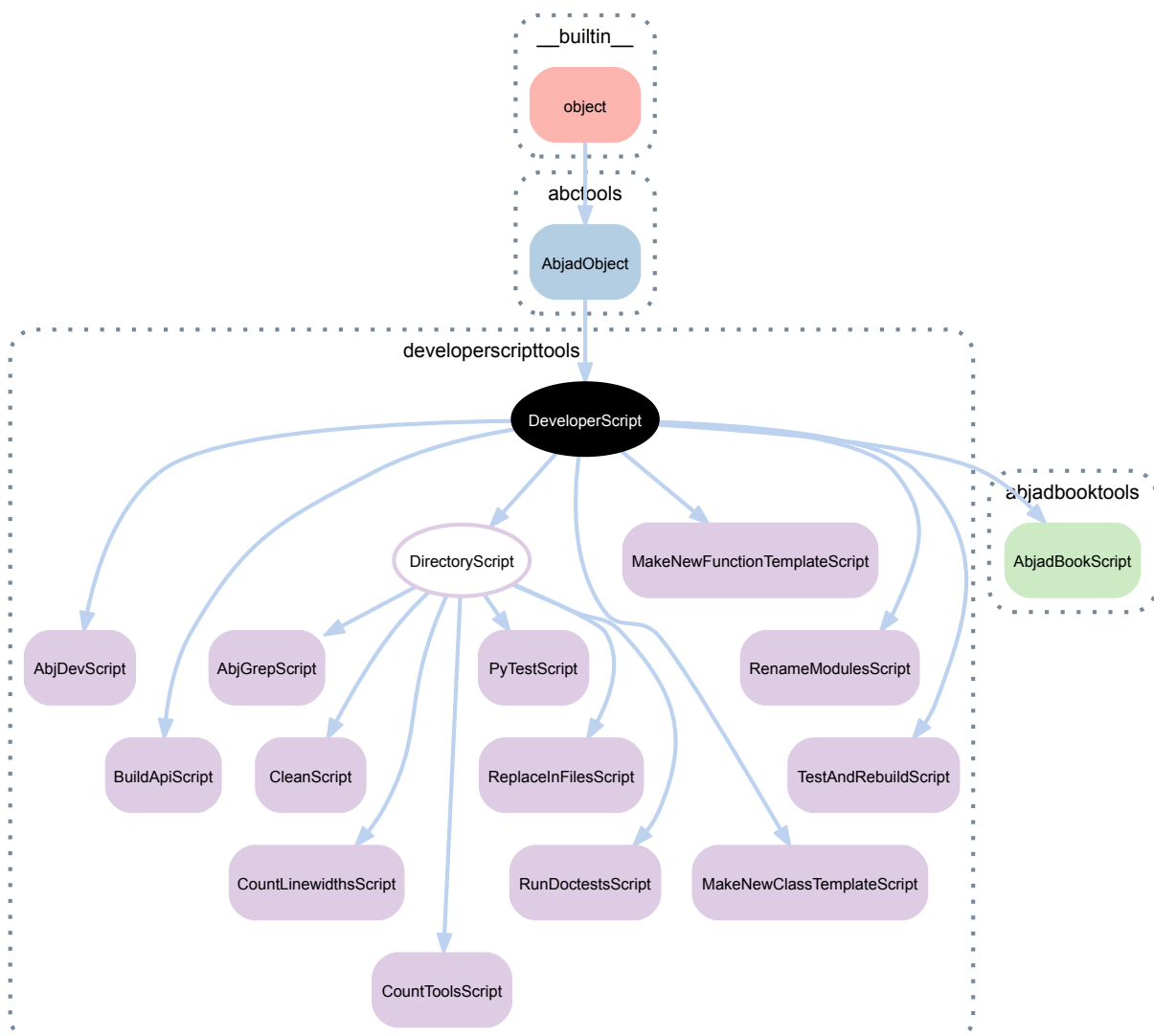
(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.1 Abstract classes

34.1.1 developerscripttools.DeveloperScript



class `developerscripttools.DeveloperScript`

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.colors`

Colors.

Returns dictionary.

`DeveloperScript.formatted_help`

Formatted help of developer script.

`DeveloperScript.formatted_usage`

Formatted usage of developer script.

`DeveloperScript.formatted_version`

Formatted version of developer script.

`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`

Scripting subcommand group of script.

`DeveloperScript.short_description`

Short description of the script, printed before arguments explanations.

Also used as a summary in other contexts.

`DeveloperScript.version`

Version number of developer script.

Methods

`DeveloperScript.process_args (args)`

Processes *args*.

`DeveloperScript.setup_argument_parser ()`

Sets up argument parser.

Special methods

`DeveloperScript.__call__ (args=None)`

Calls developer script.

Returns none.

(AbjadObject) .**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

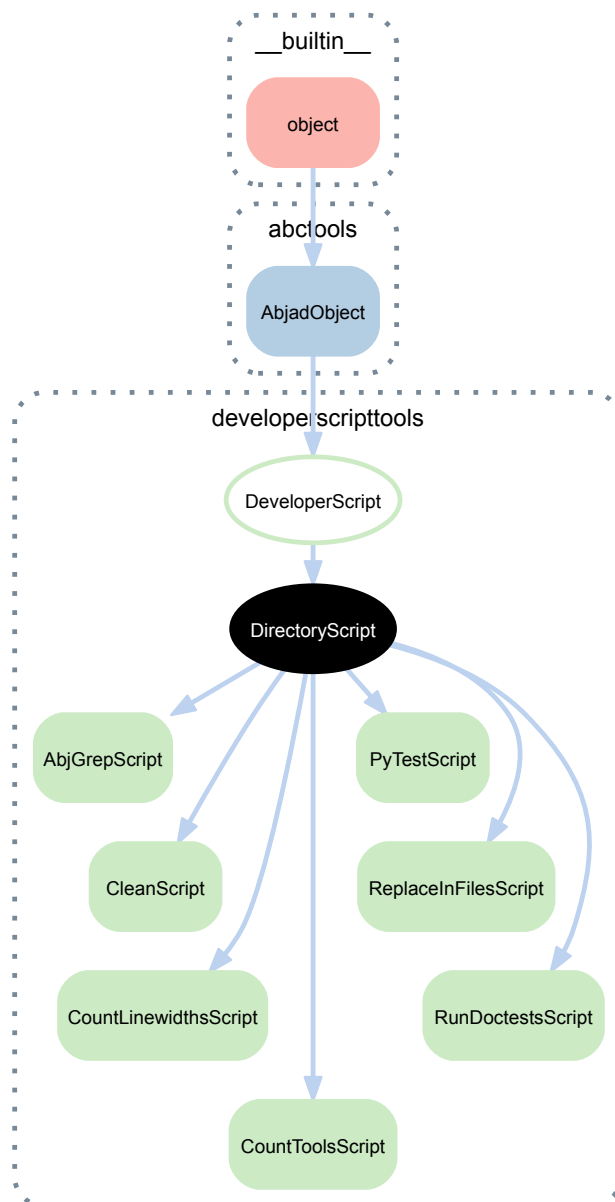
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.1.2 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).colors`
Colors.
Returns dictionary.

`(DeveloperScript).formatted_help`
Formatted help of developer script.

`(DeveloperScript).formatted_usage`
Formatted usage of developer script.

`(DeveloperScript).formatted_version`
Formatted version of developer script.

`(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`
Scripting subcommand group of script.

`(DeveloperScript).short_description`
Short description of the script, printed before arguments explanations.
Also used as a summary in other contexts.

`(DeveloperScript).version`
Version number of developer script.

Methods

`(DeveloperScript).process_args(args)`
Processes *args*.

`(DeveloperScript).setup_argument_parser()`
Sets up argument parser.

Special methods

`(DeveloperScript).__call__(args=None)`
Calls developer script.
Returns none.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

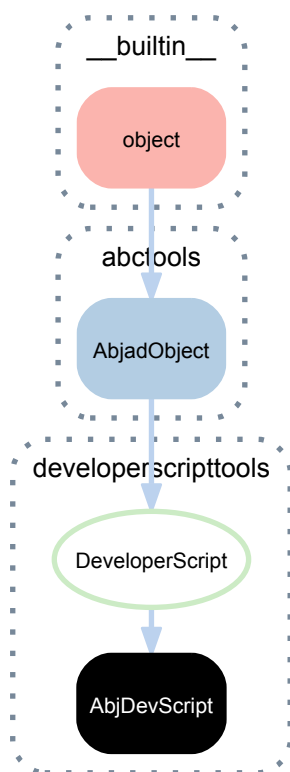
`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

34.2 Concrete classes

34.2.1 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,doctest,grep,new,re,replace,test}
                ...

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,doctest,grep,new,re,replace,test}
  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
  doctest             Run doctests on all modules in current path.
  grep                grep PATTERN in PATH
```

<code>new</code>	"new"-related subcommands
<code>re</code>	Run <code>pytest -x</code> , <code>doctest -x</code> and then rebuild the API.
<code>rename</code>	Rename public modules.
<code>replace</code>	Replace text.
<code>test</code>	Run "pytest" on various Abjad paths.

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
```

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).**colors**

Colors.

Returns dictionary.

AbjDevScript.**developer_script_aliases**

Developer script aliases.

AbjDevScript.**developer_script_classes**

Developer scripts classes.

AbjDevScript.**developer_script_program_names**

Developer script program names.

(DeveloperScript).**formatted_help**

Formatted help of developer script.

(DeveloperScript).**formatted_usage**

Formatted usage of developer script.

(DeveloperScript).**formatted_version**

Formatted version of developer script.

AbjDevScript.**long_description**

Long description.

(DeveloperScript).**program_name**

The name of the script, callable from the command line.

`(DeveloperScript).scripting_group`

Scripting subcommand group of script.

`AbjDevScript.short_description`

Short description.

`AbjDevScript.version`

Version.

Methods

`AbjDevScript.process_args (args)`

Processes *args*.

`AbjDevScript.setup_argument_parser (parser)`

Sets up argument *parser*.

Special methods

`AbjDevScript.__call__ (args=None)`

Calls script.

`(AbjadObject).__eq__ (expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__ (format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__ (expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

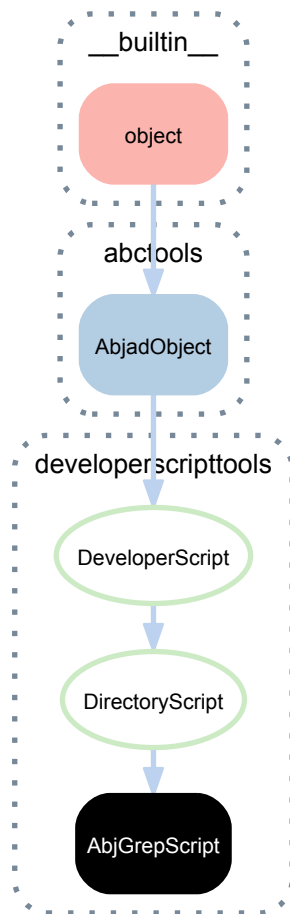
Returns boolean.

`(AbjadObject).__repr__ ()`

Gets interpreter representation of Abjad object.

Returns string.

34.2.2 developerscripttools.AbjGrepScript



class developerscripttools.AbjGrepScript

Runs *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-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.
  
```

Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`

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

Read-only properties

`AbjGrepScript.alias`

Alias of script.

Returns 'grep'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`AbjGrepScript.long_description`

Long description of script.

Returns string.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`AbjGrepScript.scripting_group`

Scripting group of script.

Returns none.

`AbjGrepScript.short_description`

Short description of script.

Returns string.

`AbjGrepScript.version`

Version of script.

Returns float.

Methods

`AbjGrepScript.process_args(args)`

Processes *args*.

Returns none.

`AbjGrepScript.setup_argument_parser(parser)`

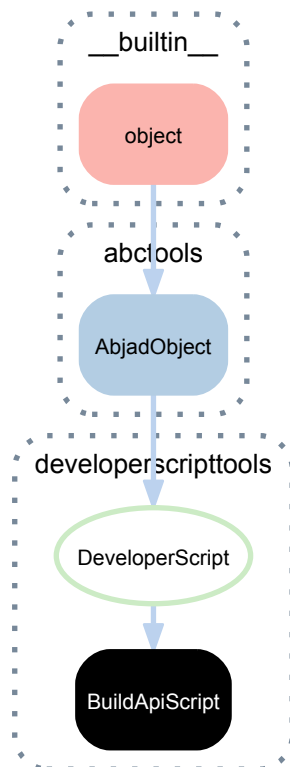
Sets up argument *parser*.

Returns none.

Special methods

- (DeveloperScript).**__call__**(args=None)
Calls developer script.
Returns none.
- (AbjadObject).**__eq__**(expr)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.
- (AbjadObject).**__format__**(format_specification='')
Formats object.
Set *format_specification* to ' ' or 'storage'. Interprets ' ' equal to 'storage'.
Returns string.
- (AbjadObject).**__ne__**(expr)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.
- (AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

34.2.3 developerscripttools.BuildApiScript



class developerscripttools.**BuildApiScript**
Builds 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
```

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`BuildApiScript`.**alias**

Alias of script.

Returns 'api'.

`(DeveloperScript)`.**argument_parser**

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript)`.**colors**

Colors.

Returns dictionary.

`(DeveloperScript)`.**formatted_help**

Formatted help of developer script.

`(DeveloperScript)`.**formatted_usage**

Formatted usage of developer script.

`(DeveloperScript)`.**formatted_version**

Formatted version of developer script.

`BuildApiScript`.**long_description**

Long description of script.

Returns string or none.

`(DeveloperScript)`.**program_name**

The name of the script, callable from the command line.

`BuildApiScript`.**scripting_group**

Scripting group of script.

Returns none.

`BuildApiScript`.**short_description**

Short description of script.

Returns string.

`BuildApiScript`.**version**

Version of script.

Returns float.

Methods

`BuildApiScript.process_args(args)`

Processes *args*.

Returns none.

`BuildApiScript.setup_argument_parser(parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`

Calls developer script.

Returns none.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

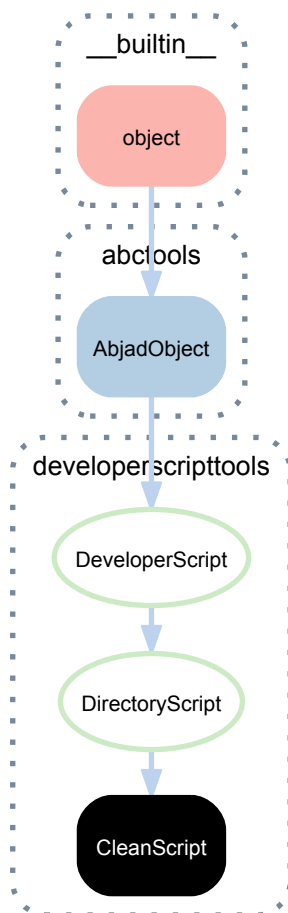
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

34.2.4 developerscripttools.CleanScript



class developerscripttools.CleanScript

Removes *.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
  
```

Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`CleanScript.alias`

Alias of script.

Returns 'clean'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`CleanScript.long_description`

Long description of scrip.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`CleanScript.scripting_group`

Scripting group of script.

Returns none.

`CleanScript.short_description`

Short description of script.

Returns string.

`CleanScript.version`

Version of script.

Returns float.

Methods

`CleanScript.process_args(args)`

Processes *args*.

Returns none.

`CleanScript.setup_argument_parser(parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`

Calls developer script.

Returns none.

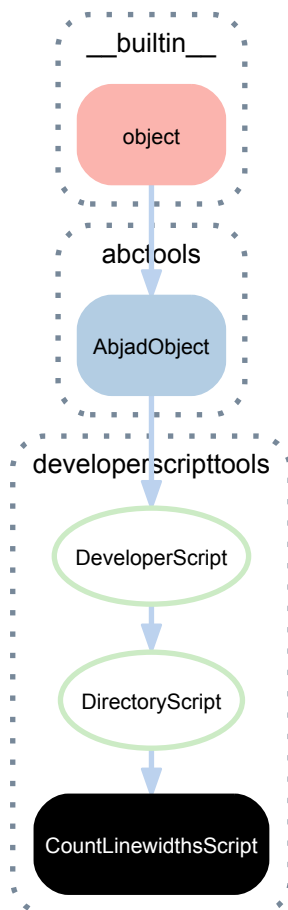
(AbjadObject) .**__eq__**(*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject) .**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject) .**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

34.2.5 developerscripttools.CountLinewidthsScript



class developerscripttools.**CountLinewidthsScript**
 Counts 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

```

Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`CountLinewidthsScript.alias`

Alias of script.

Returns 'linewidths'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`CountLinewidthsScript.long_description`

Long description of script.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`CountLinewidthsScript.scripting_group`

Scripting group of script.

Returns 'count'.

`CountLinewidthsScript.short_description`

Short description of script.

Returns string.

`CountLinewidthsScript.version`

Version of script.

Returns float.

Methods

`CountLinewidthsScript.process_args (args)`

Processes *args*.

Returns none.

`CountLinewidthsScript.setup_argument_parser (parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__ (args=None)`

Calls developer script.

Returns none.

`(AbjadObject).__eq__ (expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__ (format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__ (expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

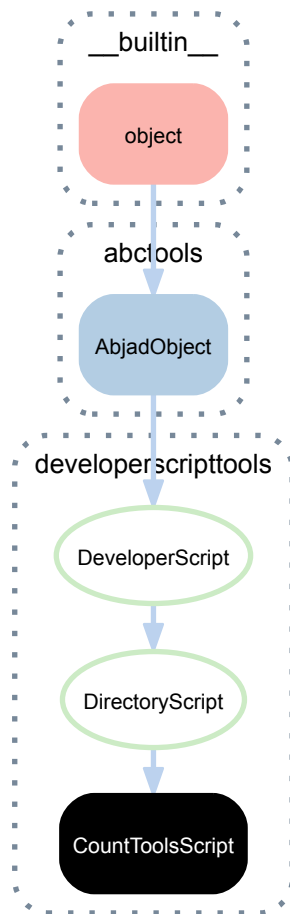
Returns boolean.

`(AbjadObject).__repr__ ()`

Gets interpreter representation of Abjad object.

Returns string.

34.2.6 developerscripttools.CountToolsScript



class `developerscripttools.CountToolsScript`
 Counts 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
  
```

Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`CountToolsScript.alias`

Alias of script.

Returns 'tools'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`CountToolsScript.long_description`

Long description of script.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`CountToolsScript.scripting_group`

Scripting group of script.

Returns 'count'.

`CountToolsScript.short_description`

Short description of script.

Returns string.

`CountToolsScript.version`

Version of script.

Returns float.

Methods

`CountToolsScript.process_args(args)`

Processes *args*.

Returns none.

`CountToolsScript.setup_argument_parser(parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`

Calls developer script.

Returns none.

(AbjadObject) .**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

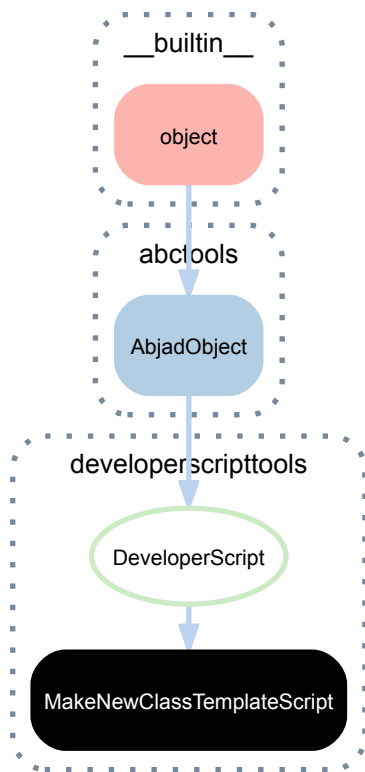
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.2.7 developerscripttools.MakeNewClassTemplateScript



class developerscripttools.**MakeNewClassTemplateScript**

Creates class stubs and 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
```

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`MakeNewClassTemplateScript.alias`

Alias of script.

Returns 'class'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`MakeNewClassTemplateScript.long_description`

Long description of script.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`MakeNewClassTemplateScript.scripting_group`

Scripting group of script.

Returns 'new'.

`MakeNewClassTemplateScript.short_description`

Short description of script.

Returns string.

`MakeNewClassTemplateScript.version`

Version of script.

Returns float.

Methods

`MakeNewClassTemplateScript.process_args(args)`

Processes *args*.

Returns none.

`MakeNewClassTemplateScript.setup_argument_parser(parser)`
 Sets up argument *parser*.
 Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`
 Calls developer script.
 Returns none.

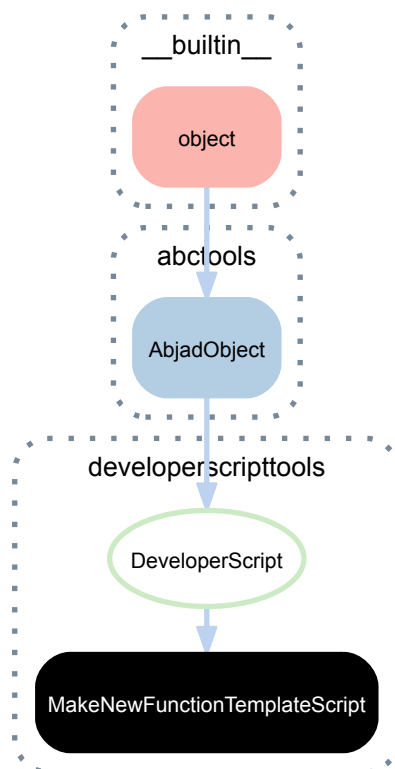
`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

34.2.8 developerscripttools.MakeNewFunctionTemplateScript



class `developerscripttools.MakeNewFunctionTemplateScript`
Makes new 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
```

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`MakeNewFunctionTemplateScript.alias`
Alias of script.

Returns 'function'.

`(DeveloperScript).argument_parser`
The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`
Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`
Formatted help of developer script.

`(DeveloperScript).formatted_usage`
Formatted usage of developer script.

`(DeveloperScript).formatted_version`
Formatted version of developer script.

`MakeNewFunctionTemplateScript.long_description`
Long description of script.

Returns string or none.

`(DeveloperScript).program_name`
The name of the script, callable from the command line.

`MakeNewFunctionTemplateScript.scripting_group`
Scripting group of script.

Returns 'new'.

`MakeNewFunctionTemplateScript.short_description`
Short description of script.

Returns string.

`MakeNewFunctionTemplateScript.version`

Version of script.

Returns float.

Methods

`MakeNewFunctionTemplateScript.process_args (args)`

Processes *args*.

Returns none.

`MakeNewFunctionTemplateScript.setup_argument_parser (parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__ (args=None)`

Calls developer script.

Returns none.

`(AbjadObject).__eq__ (expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__ (format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__ (expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

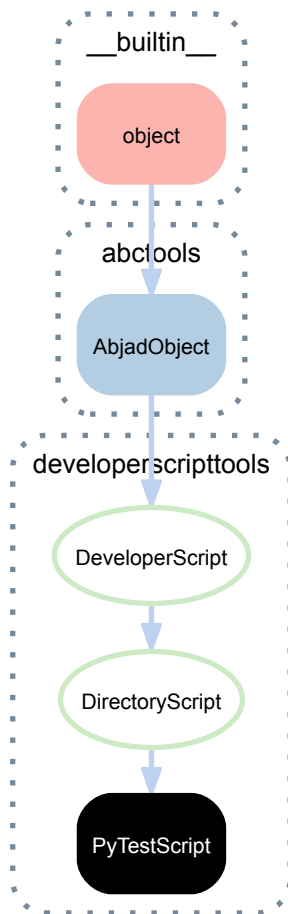
Returns boolean.

`(AbjadObject).__repr__ ()`

Gets interpreter representation of Abjad object.

Returns string.

34.2.9 developerscripttools.PyTestScript



class `developerscripttools.PyTestScript`

Runs pytest on various Abjad paths.

```

abjad$ ajv test --help
usage: py-test [-h] [--version] [-p] [-r chars] [-x] [-A | -D | -M | -X]

Run "pytest" 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 pytest 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
  
```

Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`PyTestScript.alias`

Alias of script.

Returns 'test'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`PyTestScript.long_description`

Long description of script.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`PyTestScript.scripting_group`

Scripting group of script.

Returns none.

`PyTestScript.short_description`

Short description of script.

Returns string.

`PyTestScript.version`

Version of script.

Returns float.

Methods

`PyTestScript.process_args(args)`

Processes *args*.

Returns none.

`PyTestScript.setup_argument_parser(parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`

Calls developer script.

Returns none.

(AbjadObject) .**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

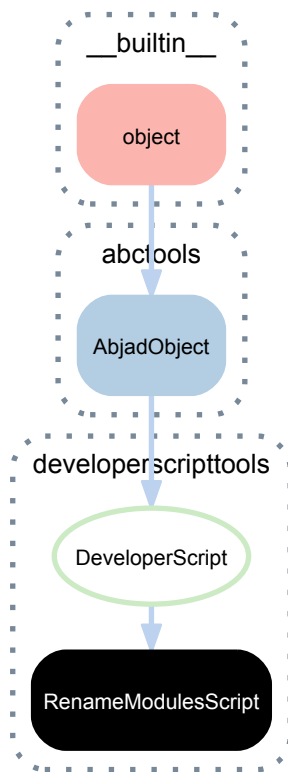
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.2.10 developerscripttools.RenameModulesScript



class developerscripttools.**RenameModulesScript**

Renames 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] source destination

Rename public modules.

positional arguments:
  source      toolspackge path of source module
  destination toolspackge path of destination module
  
```

```
optional arguments:
  -h, --help      show this help message and exit
  --version       show program's version number and exit
```

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`RenameModulesScript.alias`

Alias of script.

Returns 'rename'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`RenameModulesScript.long_description`

Long description of script.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`RenameModulesScript.scripting_group`

Scripting group of script.

Returns none.

`RenameModulesScript.short_description`

Short description of script.

Returns string.

`RenameModulesScript.version`

Version of script.

Returns float.

Methods

`RenameModulesScript.process_args(args)`

Processes *args*.

Returns none.

`RenameModulesScript.setup_argument_parser(parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`

Calls developer script.

Returns none.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

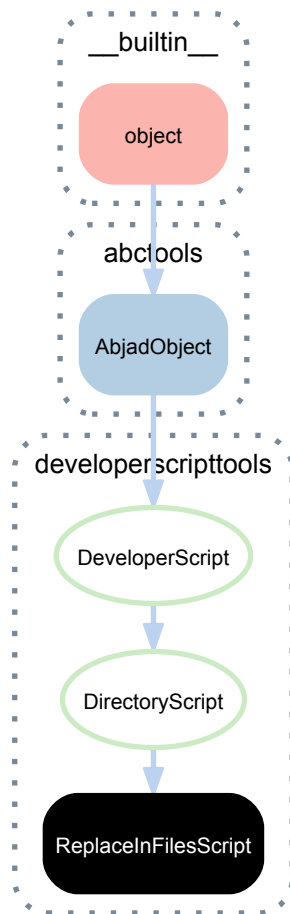
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

34.2.11 developerscripttools.ReplaceInFilesScript



class `developerscripttools.ReplaceInFilesScript`

Replaces 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
```

Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ReplaceInFilesScript.alias`

Alias of script.

Returns 'replace'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`ReplaceInFilesScript.long_description`

Long description of script.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`ReplaceInFilesScript.scripting_group`

Scripting group of script.

Returns none.

`ReplaceInFilesScript.short_description`

Short description.

Returns string.

`ReplaceInFilesScript.skipped_directories`

Skipped directories.

Returns list.

`ReplaceInFilesScript.skipped_files`

Skipped files.

Returns list.

`ReplaceInFilesScript.version`

Version of script.

Returns float.

Methods

`ReplaceInFilesScript.process_args(args)`

Processes *args*.

Returns none.

`ReplaceInFilesScript.setup_argument_parser(parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`

Calls developer script.

Returns none.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

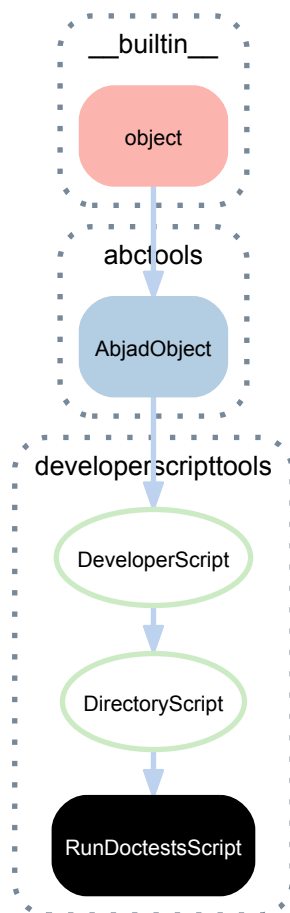
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

34.2.12 developerscripttools.RunDoctestsScript



class `developerscripttools.RunDoctestsScript`
 Runs doctests on all Python files in current directory recursively.

```

abjad$ ajv doctest --help
usage: run-doctests [-h] [--version] [--diff] [path]

Run doctests on all modules in current 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
  --diff              print diff-like output on failed tests.
  
```

Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`RunDoctestsScript.alias`

Alias of script.

Returns 'doctest'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`RunDoctestsScript.long_description`

Long description of script.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`RunDoctestsScript.scripting_group`

Scripting group of script.

Returns none.

`RunDoctestsScript.short_description`

Short description of script.

Returns string.

`RunDoctestsScript.version`

Version of script.

Returns float.

Methods

`RunDoctestsScript.process_args(args)`

Processes *args*.

Returns none.

`RunDoctestsScript.setup_argument_parser(parser)`

Sets up argument *parser*.

Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`

Calls developer script.

Returns none.

(AbjadObject) .**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

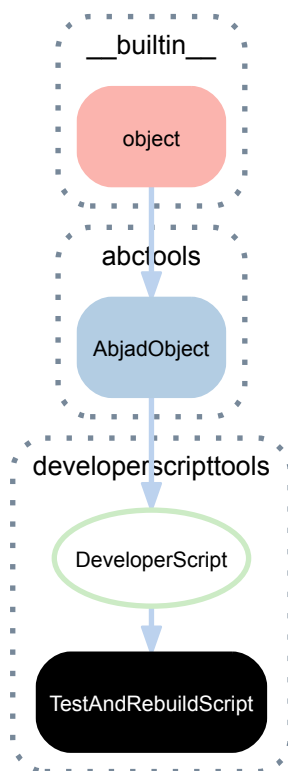
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.2.13 developerscripttools.TestAndRebuildScript



class developerscripttools.**TestAndRebuildScript**

Tests codebase and rebuilds docs.

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`TestAndRebuildScript.alias`

Alias of script.

Returns 're'.

`(DeveloperScript).argument_parser`

The script's instance of `argparse.ArgumentParser`.

`(DeveloperScript).colors`

Colors.

Returns dictionary.

`(DeveloperScript).formatted_help`

Formatted help of developer script.

`(DeveloperScript).formatted_usage`

Formatted usage of developer script.

`(DeveloperScript).formatted_version`

Formatted version of developer script.

`TestAndRebuildScript.long_description`

Long description of script.

Returns string or none.

`(DeveloperScript).program_name`

The name of the script, callable from the command line.

`TestAndRebuildScript.scripting_group`

Scripting group of script.

Returns none.

`TestAndRebuildScript.short_description`

Short description of script.

Returns string.

`TestAndRebuildScript.version`

Version of script.

Returns float.

Methods

`TestAndRebuildScript.get_terminal_width()`

Borrowed from the py lib.

`TestAndRebuildScript.process_args(args)`

Processes *args*.

Returns none.

`TestAndRebuildScript.rebuild_docs(args)`

Rebuilds docs.

`TestAndRebuildScript.run_doctest(args)`

Runs doctest.

`TestAndRebuildScript.run_pytest(args)`

Runs pytest.

`TestAndRebuildScript.setup_argument_parser(parser)`
Sets up argument *parser*.
Returns none.

Special methods

`(DeveloperScript).__call__(args=None)`
Calls developer script.
Returns none.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

34.3 Functions

34.3.1 developerscripttools.get_developer_script_classes

`developerscripttools.get_developer_script_classes()`
Returns a list of all developer script classes.

34.3.2 developerscripttools.run_abjadbook

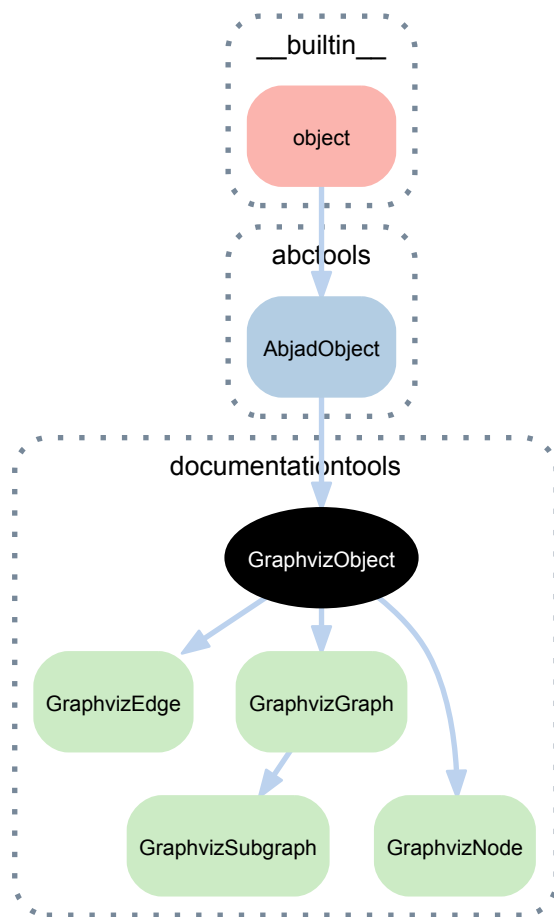
`developerscripttools.run_abjadbook()`
Entry point for `setuptools`.
One-line wrapper around `AbjadBookScript`.

34.3.3 developerscripttools.run_ajv

`developerscripttools.run_ajv()`
Entry point for `setuptools`.
One-line wrapper around `AbjDevScript`.

35.1 Abstract classes

35.1.1 documentationtools.GraphvizObject



class `documentationtools.GraphvizObject` (*attributes=None*)
An attributed Graphviz object.

Bases

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

Read-only properties

`GraphvizObject.attributes`
Attributes of Graphviz object.

Special methods

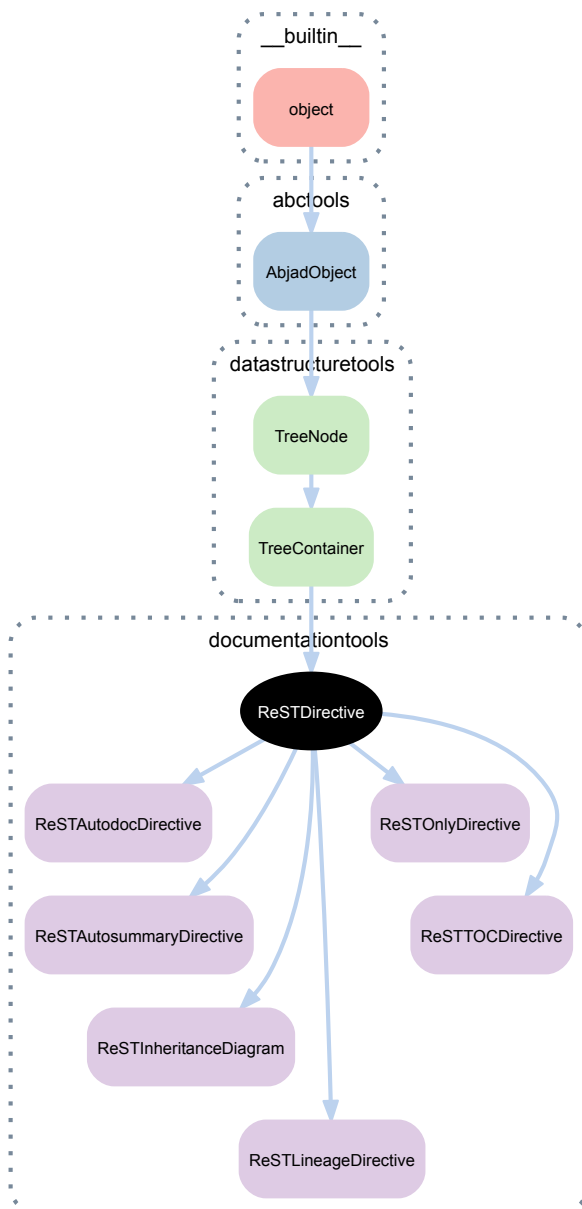
`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

35.1.2 documentationtools.ReSTDirective



class `documentationtools.ReSTDirective` (*argument=None, children=None, name=None, options=None*)
 A ReST directive.

Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(TreeContainer).children`
 Children of tree container.

```
>>> 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 tree nodes.

(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`

Gets and sets directive of ReST directive.

`(TreeNode).graph_order`

Graph order of tree node.

Returns tuple.

`(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 tree nodes.

`(TreeContainer).leaves`

Leaves of tree container.

```
>>> 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`

Node class of ReST directive.

`(TreeContainer).nodes`

The collection of tree nodes produced by iterating tree container 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
Options of ReST directive.

(TreeNode).parent
Parent of tree 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
```

Returns tree node.

(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 tree nodes.

`ReSTDirective.rest_format`

ReST format of ReST directive.

`(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
```

Returns tree node.

Read/write properties

`ReSTDirective.argument`

Gets and sets argument of ReST directive.

`(TreeNode).name`

Named of tree node.

Returns string.

Methods

`(TreeContainer).append(node)`

Appends *node* to tree 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(),
  )
)
```

Returns none.

`(TreeContainer).extend(expr)`

Extendes *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)

Indexes *node* in tree 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 tree 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*)

Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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)
Copies tree node.

Returns new tree node.

(TreeNode) .**__deepcopy__** (*args)
Copies tree node.

Returns new tree node.

(TreeContainer) .**__delitem__** (i)
Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.
Returns boolean.

(AbjadObject) .**__format__** (format_specification='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(TreeContainer) .**__getitem__** (i)
Gets node *i* in tree 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.

(TreeContainer).**__iter__**()
Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()
Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)
True when tree node does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(TreeContainer).**__setitem__**(*i*, *expr*)
Sets *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
```

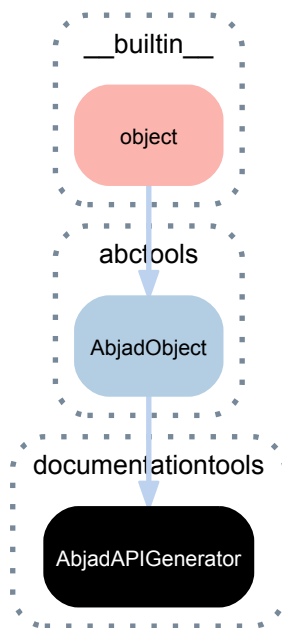
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

35.2 Concrete classes

35.2.1 documentationtools.AbjadAPIGenerator



class `documentationtools.AbjadAPIGenerator`

Generates the Abjad API restructured text.

- 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

Bases

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

Read-only properties

`AbjadAPIGenerator.docs_api_index_path`

Path to index.rst for Abjad API.

`AbjadAPIGenerator.package_prefix`

Package prefix.

`AbjadAPIGenerator.path_definitions`

Code path / docs path / package prefix triples.

`AbjadAPIGenerator.root_package`

Root package.

Returns 'abjad'.

`AbjadAPIGenerator.tools_package_path_index`

Tools package path index.

Returns 2.

Special methods

`AbjadAPIGenerator.__call__(verbose=False)`

Calls Abjad API generator.

Returns none.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

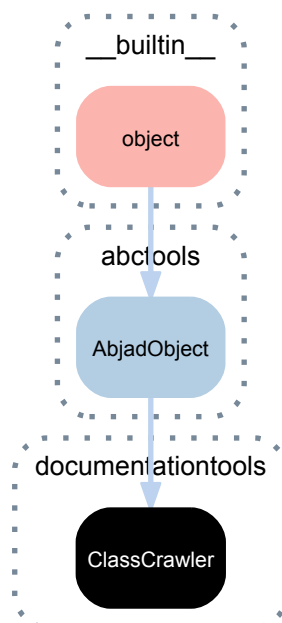
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

35.2.2 documentationtools.ClassCrawler



class `documentationtools.ClassCrawler` (*code_root=None*, *include_private_objects=False*,
root_package_name=None)
Class crawler.

Bases

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

Read-only properties

`ClassCrawler.code_root`
Code root of class crawler.

`ClassCrawler.include_private_objects`
Include private objects.

`ClassCrawler.module_crawler`
Module crawler.

`ClassCrawler.root_package_name`
Root package name.

Special methods

`ClassCrawler.__call__()`
Calls class crawler.
Returns tuple of classes.

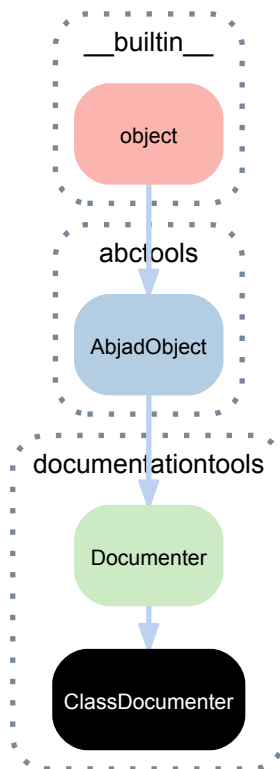
`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to `''` or `'storage'`. Interprets `''` equal to `'storage'`.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

35.2.3 documentationtools.ClassDocumenter



class `documentationtools.ClassDocumenter` (*object_=None, 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

.. autoclass:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter

.. only:: html

Attribute summary
-----

.. autosummary::

    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.class_methods
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.data
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.inherited_attributes
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.is_abstract
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.methods
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.module_name
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.object_
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.prefix
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.readonly_properties
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.readwrite_properties
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.special_methods
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.static_methods
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.write
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__call__
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__eq__
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__makenew__
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__ne__
    ~abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__repr__
  
```

```

Bases
-----

- :py:class:`documentationtools.Documenter` <abjad.tools.documentationtools.Documenter.Documenter>`
- :py:class:`abctools.AbjadObject` <abjad.tools.abctools.AbjadObject.AbjadObject>`
- :py:class:`__builtin__.object` <object>`

Read-only properties
-----

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.class_methods
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.data
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.inherited_attributes
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.is_abstract
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.methods
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.module_name
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.object_
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.prefix
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.readonly_properties
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.readwrite_properties
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.special_methods
   :noindex:

.. autoattribute:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.static_methods
   :noindex:

Static methods
-----

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.write
   :noindex:

Special methods
-----

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__call__
   :noindex:

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__eq__
   :noindex:

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__makenew__
   :noindex:

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__ne__
   :noindex:

.. automethod:: abjad.tools.documentationtools.ClassDocumenter.ClassDocumenter.__repr__
   :noindex:

```

Bases

- `documentationtools.Documenter`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ClassDocumenter.class_methods`
Class methods.

`ClassDocumenter.data`
Data.

`ClassDocumenter.inherited_attributes`
Inherited attributes.

`ClassDocumenter.is_abstract`
True when class is abstract. Otherwise false.

Returns boolean.

`ClassDocumenter.methods`
Methods of class.

`(Documenter).module_name`
Module name of documenter.

Returns string.

`(Documenter).object_`
Object of documenter.

`(Documenter).prefix`
Prefix of documenter.

`ClassDocumenter.readonly_properties`
Read-only properties of class.

`ClassDocumenter.readwrite_properties`
The read / write properties of class.

`ClassDocumenter.special_methods`
Special methods of class.

`ClassDocumenter.static_methods`
Static methods of class.

Static methods

`(Documenter).write(file_path, restructured_text)`
Writes *restructured_text* to *file_path*.

Returns none.

Special methods

`ClassDocumenter.__call__()`
Calls class documenter.

Generates documentation.

Returns string.

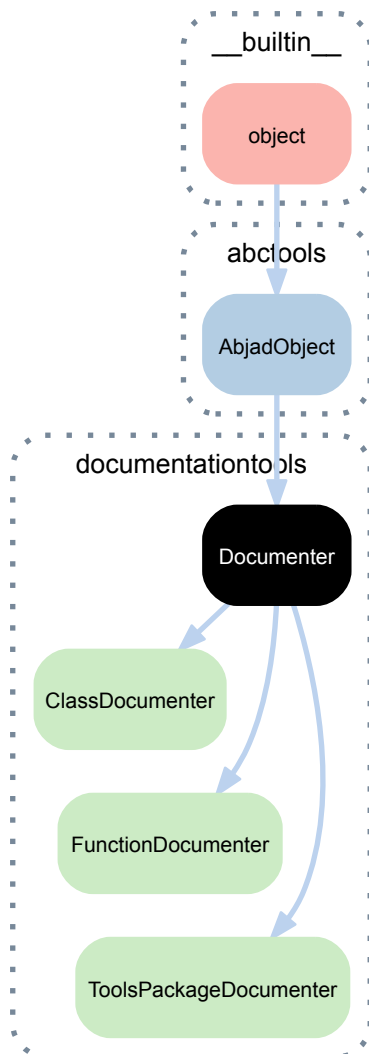
(AbjadObject) .**__eq__**(*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(Documenter) .**__makenew__**(*object_=None, prefix=None*)
 Makes new documenter.
 Returns new documenter.

(AbjadObject) .**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject) .**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

35.2.4 documentationtools.Documenter



class documentationtools.**Documenter**(*object_, prefix='abjad.tools.'*)
 Documenter is an abstract base class for documentation classes.

Bases

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

Read-only properties

`Documenter.module_name`
Module name of documenter.

Returns string.

`Documenter.object_`
Object of documenter.

`Documenter.prefix`
Prefix of documenter.

Static methods

`Documenter.write` (*file_path*, *restructured_text*)
Writes *restructured_text* to *file_path*.

Returns none.

Special methods

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`Documenter.__makenew__` (*object_=None*, *prefix=None*)
Makes new documenter.

Returns new documenter.

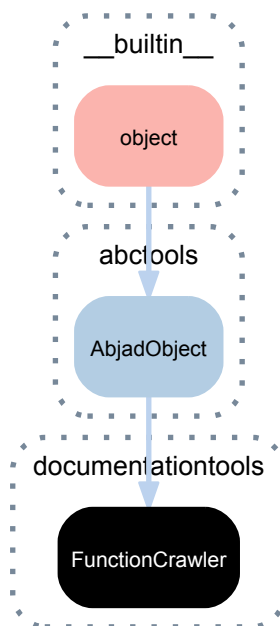
`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__` ()
Gets interpreter representation of Abjad object.

Returns string.

35.2.5 documentationtools.FunctionCrawler



class `documentationtools.FunctionCrawler` (*code_root=None*, *include_private_objects=False*, *root_package_name=None*) *in-*

Function crawler.

Bases

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

Read-only properties

`FunctionCrawler.code_root`
Code root of function crawler.

`FunctionCrawler.include_private_objects`
Include private objects.

`FunctionCrawler.module_crawler`
Module crawler.

`FunctionCrawler.root_package_name`
Root package name.

Special methods

`FunctionCrawler.__call__()`
Calls function crawler.

Returns tuple of functions.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

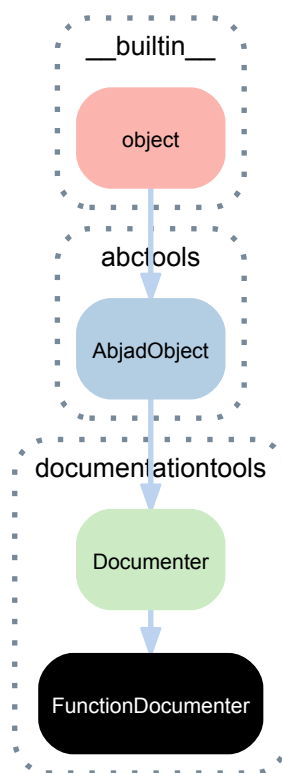
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

35.2.6 documentationtools.FunctionDocumenter



class documentationtools.**FunctionDocumenter**(*object_=None*, *prefix='abjad.tools.'*)

FunctionDocumenter generates an ReST entry for a given function:

```

>>> documenter = documentationtools.FunctionDocumenter(scoretools.make_notes)
>>> print documenter()
scoretools.make_notes
=====
.. autofunction:: abjad.tools.scoretools.make_notes.make_notes
  
```

Returns FunctionDocumenter `instance`.

Bases

- documentationtools.Documenter
- abctools.AbjadObject

- `__builtin__.object`

Read-only properties

`(Documenter).module_name`
Module name of documenter.

Returns string.

`(Documenter).object_`
Object of documenter.

`(Documenter).prefix`
Prefix of documenter.

Static methods

`(Documenter).write(file_path, restructured_text)`
Writes *restructured_text* to *file_path*.

Returns none.

Special methods

`FunctionDocumenter.__call__()`
Generate documentation.

Returns string.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(Documenter).__makenew__(object_=None, prefix=None)`
Makes new documenter.

Returns new documenter.

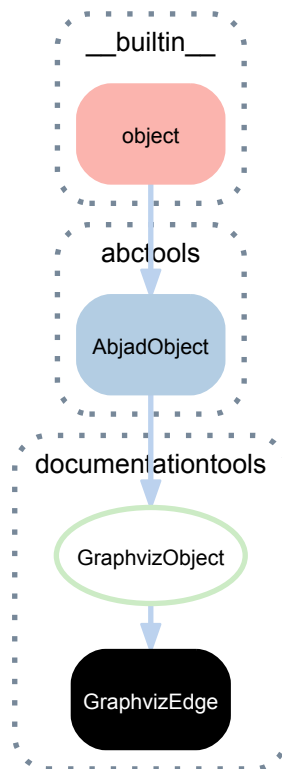
`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`
Gets interpreter representation of Abjad object.

Returns string.

35.2.7 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`
 Attributes of Graphviz object.

`GraphvizEdge.head`
 Head of Graphviz edge.

`GraphvizEdge.tail`
 Tail of Graphviz edge.

Read/write properties

`GraphvizEdge.is_directed`
 True when Graphviz edge is directed. Otherwise false.
 Returns boolean.

Special methods

`GraphvizEdge.__call__(*args)`

Calls Graphviz edge.

Returns Graphviz edge.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

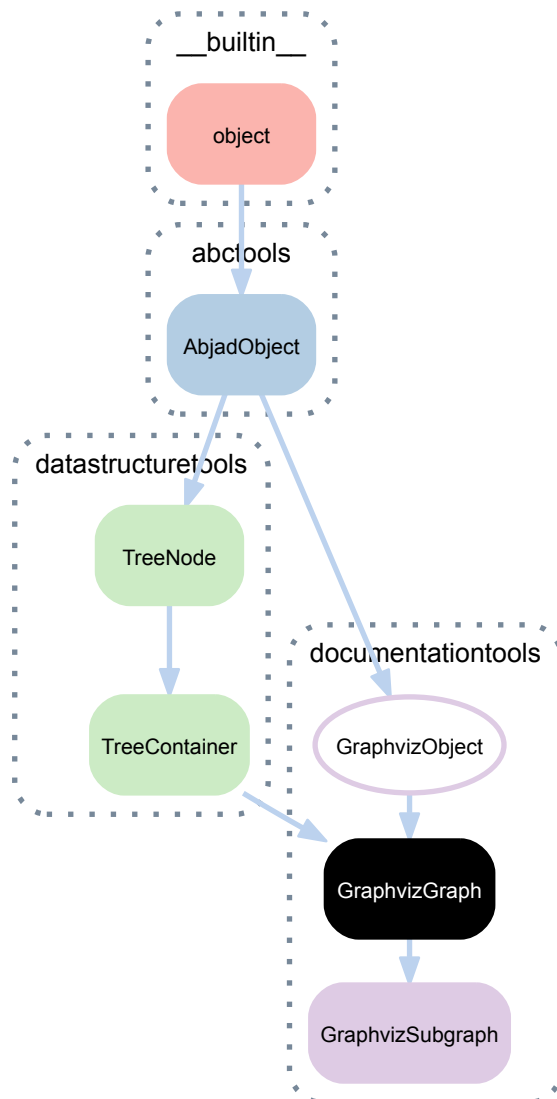
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

35.2.8 documentationtools.GraphvizGraph



class `documentationtools.GraphvizGraph` (*attributes=None*, *children=None*,
edge_attributes=None, *is_digraph=True*,
name=None, *node_attributes=None*)

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'
```

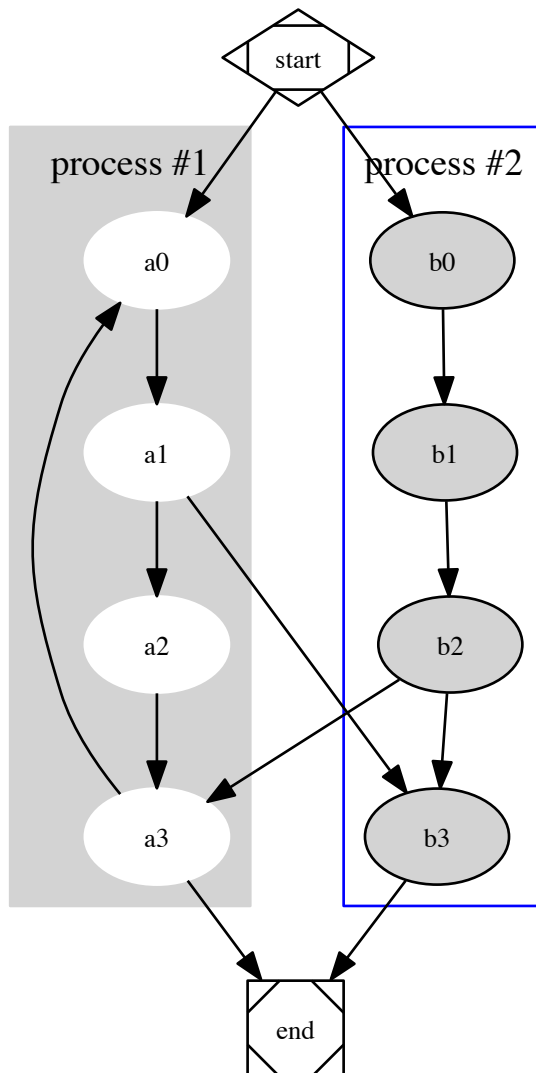
Access the computed graphviz format of the graph:

```
>>> 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:

```
>>> topleveltools.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;
}

```

Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `documentationtools.GraphvizObject`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(`GraphvizObject`).**.attributes**
Attributes of Graphviz object.

`GraphvizGraph`.**.canonical_name**
Canonical name of Graphviz graph.
Returns string.

(`TreeContainer`).**.children**
Children of tree container.

```

>>> 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 tree nodes.

(`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**

Edge attributes of Graphviz graph.

(TreeNode). **graph_order**

Graph order of tree node.

Returns tuple.

GraphvizGraph. **graphviz_format**

Graphviz format of Graphviz graph.

Returns string.

(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 tree nodes.

(TreeContainer).**.leaves**

Leaves of tree container.

```
>>> 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**

Node attributes of Graphviz graph.

(TreeContainer).**.nodes**

The collection of tree nodes produced by iterating tree container 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**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(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
```

Returns tree node.

GraphvizGraph **.unflattened_graphviz_format**

Unflattened Graphviz format of Graphviz graph.

Returns list.

Read/write properties

GraphvizGraph **.is_digraph**

True when Graphviz graph is a digraph. Otherwise false.

Returns boolean.

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

(TreeContainer) **.append** (*node*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) **.extend** (*expr*)

Extendes *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)

Indexes *node* in tree 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 tree 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*)
 Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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)
Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(*args)
Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TreeContainer).**__getitem__**(*i*)

Gets node *i* in tree 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.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

True when tree node does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TreeContainer).**__setitem__**(*i*, *expr*)

Sets *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
```

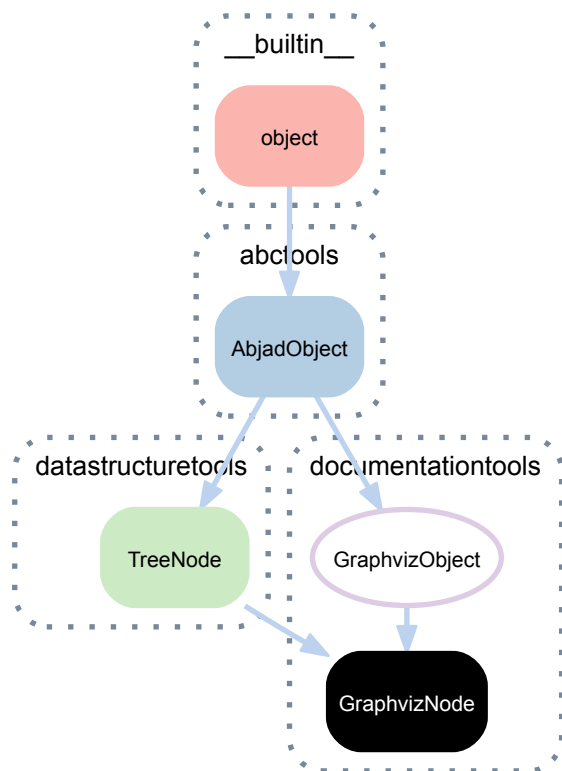
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

35.2.9 documentationtools.GraphvizNode



class `documentationtools.GraphvizNode` (*attributes=None, name=None*)
 A Graphviz node.

Bases

- `datastructuretools.TreeNode`
- `documentationtools.GraphvizObject`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(GraphvizObject).attributes`
 Attributes of Graphviz object.

`GraphvizNode.canonical_name`
 Canonical name of Graphviz node.
 Returns string.

`(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**

Edges of Graphviz node.

Returns tuple.

(TreeNode) **.graph_order**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(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
```

Returns tree node.

Read/write properties

(TreeNode) . **name**
Named of tree node.
Returns string.

Special methods

(TreeNode) . **__copy__** (*args)
Copies tree node.
Returns new tree node.

(TreeNode) . **__deepcopy__** (*args)
Copies tree node.
Returns new tree node.

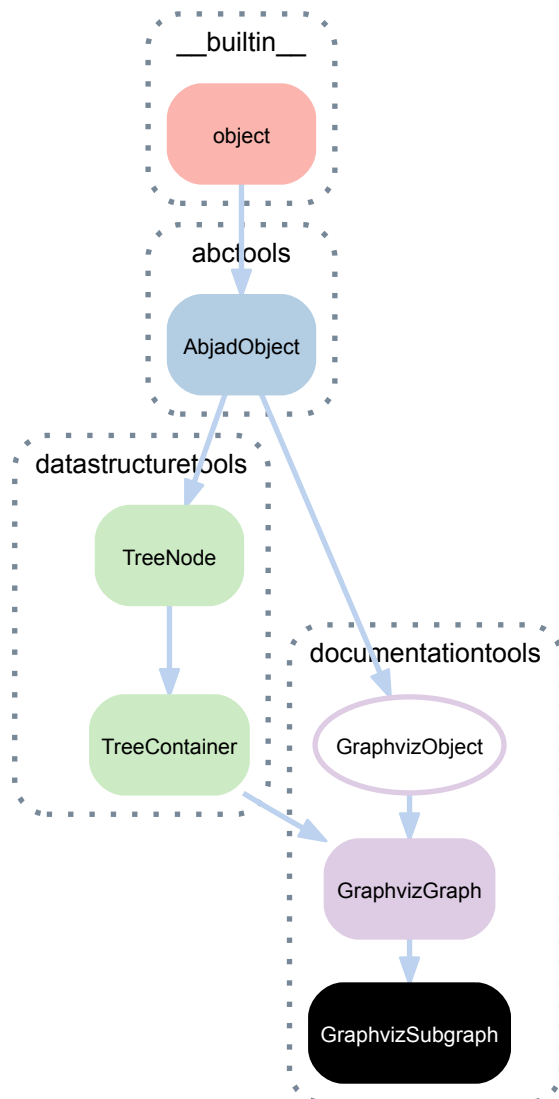
(TreeNode) . **__eq__** (expr)
True when *expr* is a tree node. Otherwise false.
Returns boolean.

(AbjadObject) . **__format__** (format_specification='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(TreeNode) . **__ne__** (expr)
True when tree node does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject) . **__repr__** ()
Gets interpreter representation of Abjad object.
Returns string.

35.2.10 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**

Attributes of Graphviz object.

GraphvizSubgraph **.canonical_name**

Canonical name of Graphviz subgraph.

Returns string.

(TreeContainer) **.children**

Children of tree container.

```
>>> 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 tree nodes.

(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**
Edge attributes of Graphviz graph.

GraphvizSubgraph **.edges**
Edges of Graphviz subgraph.

Returns tuple.

(TreeNode) **.graph_order**
Graph order of tree node.

Returns tuple.

(GraphvizGraph) **.graphviz_format**
Graphviz format of Graphviz graph.

Returns string.

(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 tree nodes.

(TreeContainer) **.leaves**
Leaves of tree container.

```

>>> 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**

Node attributes of Graphviz graph.

(TreeContainer) **.nodes**

The collection of tree nodes produced by iterating tree container 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**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(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
```

Returns tree node.

(GraphvizGraph) **.unflattened_graphviz_format**

Unflattened Graphviz format of Graphviz graph.

Returns list.

Read/write properties

GraphvizSubgraph **.is_cluster**

True when Graphviz subgraph is a cluster. Otherwise false.

Returns boolean.

(GraphvizGraph) **.is_digraph**

True when Graphviz graph is a digraph. Otherwise false.

Returns boolean.

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

(TreeContainer) **.append(*node*)**

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) **.extend** (*expr*)
 Extends *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)
 Indexes *node* in tree 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 tree 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)
Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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)
Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(*args)
Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.
Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TreeContainer).**__getitem__**(*i*)

Gets node *i* in tree 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.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

True when tree node does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TreeContainer).**__setitem__**(*i, expr*)

Sets *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
```

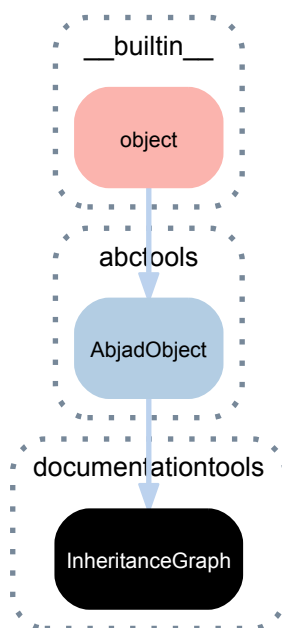
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

35.2.11 documentationtools.InheritanceGraph



```
class documentationtools.InheritanceGraph (addresses=('abjad', ), lin-
                                         eage_addresses=None, lin-
                                         eage_prune_distance=None, re-
                                         curse_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 scoretools, use this formulation:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',),
...     lineage_addresses=(scoretools,))
...     )
```

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=(instrumenttools.Instrument,),
...     lineage_prune_distance=1,
...     )
```

Returns InheritanceGraph instance.

Bases

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

Read-only properties

InheritanceGraph.**addresses**

Addresses of inheritance graph.

InheritanceGraph.**child_parents_mapping**

Child / parent mapping of inheritance graph.

`InheritanceGraph.graphviz_format`
 Graphviz format of inheritance graph.

`InheritanceGraph.graphviz_graph`
 Graphviz graph of inheritance graph.

`InheritanceGraph.immediate_classes`
 Immediate classes of inheritance graph.

`InheritanceGraph.lineage_addresses`
 Lineage addresses of inheritance graph.

`InheritanceGraph.lineage_classes`
 Lineage classes of inheritance graph.

`InheritanceGraph.lineage_distance_mapping`
 Lineage distance mapping of inheritance graph.

`InheritanceGraph.lineage_prune_distance`
 Lineage prune distance of inheritance graph.

`InheritanceGraph.parent_children_mapping`
 Parent / children mapping of inheritance graph.

`InheritanceGraph.recurse_into_submodules`
 Recurse into submodules.

`InheritanceGraph.root_addresses`
 Root addresses of inheritance graph.

`InheritanceGraph.root_classes`
 Root classes of inheritance graph.

`InheritanceGraph.use_clusters`
 Use clusters.

`InheritanceGraph.use_groups`
 Use groups.

Special methods

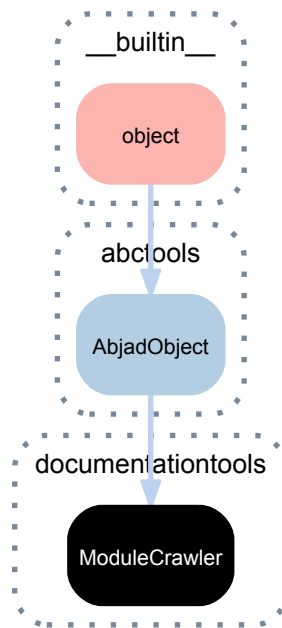
`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

35.2.12 documentationtools.ModuleCrawler



```

class documentationtools.ModuleCrawler (code_root=None,
                                         ignored_directory_names=(('__pycache__',
                                                                    '.git',
                                                                    '.svn',
                                                                    'test'),
                                                                    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`
Code root of module crawler.

Returns string.

`ModuleCrawler.ignored_directory_names`
Ignored directory names of module crawler.

Returns tuple.

`ModuleCrawler.root_package_name`
Root package name of module crawler.

Returns string.

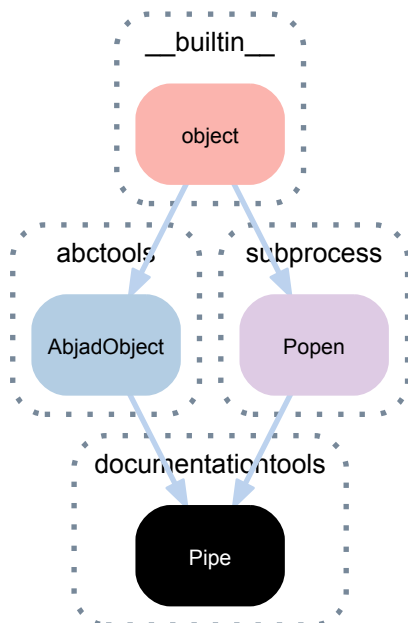
`ModuleCrawler.visit_private_modules`
Visit private modules.

Returns boolean.

Special methods

- (AbjadObject).**__eq__**(*expr*)
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.
- (AbjadObject).**__format__**(*format_specification*=')'
Formats object.
Set *format_specification* to ' or 'storage'. Interprets ' equal to 'storage'.
Returns string.
- ModuleCrawler.**__iter__**()
Iterates module crawler.
Returns generator.
- (AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.
- (AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

35.2.13 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`
Arguments of pipe.

`Pipe.executable`
Executable of pipe.

`Pipe.timeout`
Timeout of pipe.

Methods

`Pipe.close()`
Closes 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()`
Reads from pipe.

`Pipe.read_wait(seconds=0.01)`
Tries to read from pipe. Wait *seconds* if nothing comes out and the repeats.

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)`
Writes *data* into pipe.

`Pipe.write_line(data)`
Write *data* into pipe. Then writes newline to pipe.

Special methods

`(Popen).__del__(_maxint=9223372036854775807, _active=[])`

(AbjadObject) .**__eq__**(*expr*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

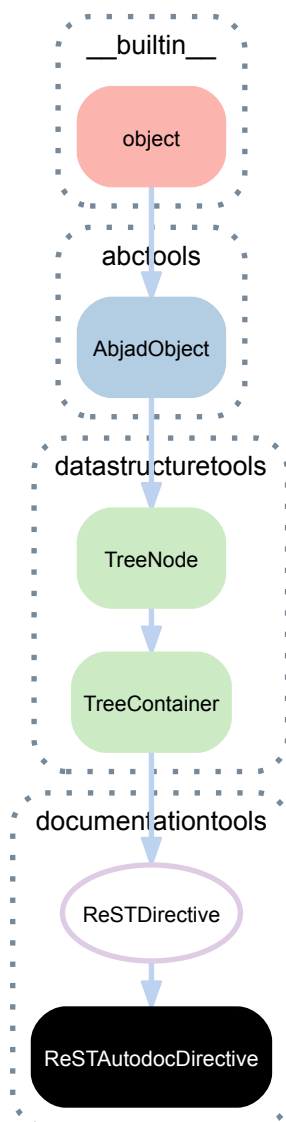
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

35.2.14 documentationtools.ReSTAutodocDirective



class `documentationtools.ReSTAutodocDirective` (*argument=None, children=None, directive='automodule', name=None, options=None*)

A ReST autodoc directive.

```
>>> autodoc = documentationtools.ReSTAutodocDirective(
...     argument='abjad.tools.spannertools.Beam.Beam',
...     directive='autoclass',
... )
>>> autodoc.options['noindex'] = True
>>> autodoc
ReSTAutodocDirective(
  argument='abjad.tools.spannertools.Beam.Beam',
  directive='autoclass',
  options={
    'noindex': True,
  }
)
```

```
>>> print autodoc.rest_format
.. autoclass:: abjad.tools.spannertools.Beam.Beam
   :noindex:
```

Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(`TreeContainer`) **.children**

Children of tree container.

```
>>> 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 tree nodes.

(`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**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeContainer) **.leaves**

Leaves of tree container.

```
>>> 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**

Node class of ReST directive.

(TreeContainer) **.nodes**

The collection of tree nodes produced by iterating tree container 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**

Options of ReST directive.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(ReSTDirective) **.rest_format**

ReST format of ReST directive.

(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
```

Returns tree node.

Read/write properties

(ReSTDirective) **.argument**

Gets and sets argument of ReST directive.

ReSTAutodocDirective **.directive**

Gets and set directive of ReST autodoc directive.

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

(TreeContainer) **.append** (*node*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) **.extend** (*expr*)

Extendes *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)

Indexes *node* in tree 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 tree 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*)
 Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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)
Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(*args)
Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.

Returns boolean.

(AbjadObject). **__format__** (*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TreeContainer). **__getitem__** (*i*)

Gets node *i* in tree 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.

(TreeContainer). **__iter__** ()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

True when tree node does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TreeContainer).**__setitem__**(*i*, *expr*)

Sets *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
```

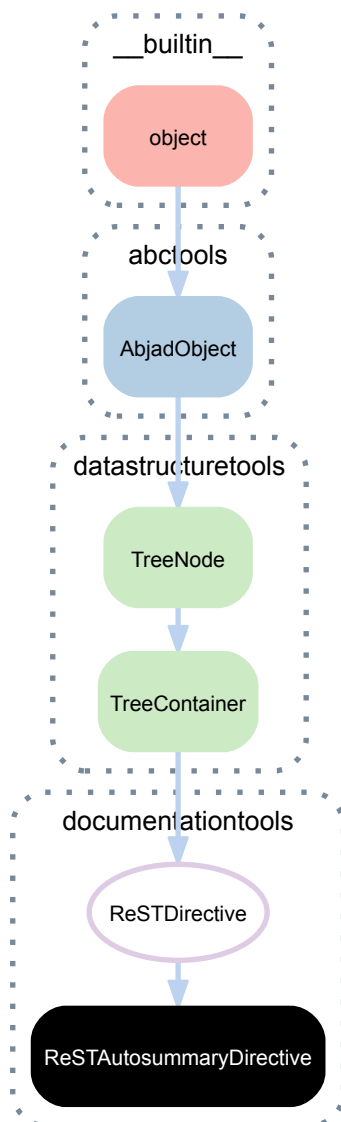
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

35.2.15 documentationtools.ReSTAutosummaryDirective



class `documentationtools.ReSTAutosummaryDirective` (*argument=None, children=None, name=None, options=None*)

A ReST Autosummary 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
```

Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(TreeContainer).children`

Children of tree container.

```
>>> 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 tree nodes.

`(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`

Directive of ReST autosummary directive.

Returns 'autosummary'.

`(TreeNode).graph_order`

Graph order of tree node.

Returns tuple.

`(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 tree nodes.

`(TreeContainer).leaves`

Leaves of tree container.

```
>>> 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`

Node class of ReST autosummary directive.

`(TreeContainer).nodes`

The collection of tree nodes produced by iterating tree container 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`

Options of ReST directive.

`(TreeNode).parent`

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(ReSTDirective) **.rest_format**

ReST format of ReST directive.

(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
```

Returns tree node.

Read/write properties

(ReSTDirective) **.argument**

Gets and sets argument of ReST directive.

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

(TreeContainer) **.append** (*node*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) **.extend** (*expr*)
 Extends *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)
 Indexes *node* in tree 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 tree 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*)
Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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*)
Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(**args*)
Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__** (*format_specification*='')
 Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TreeContainer) .**__getitem__** (*i*)
 Gets node *i* in tree 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.

(TreeContainer) .**__iter__** ()
 Iterates tree container.

Yields children of tree container.

(TreeContainer) .**__len__** ()
 Returns nonnegative integer number of nodes in container.

(TreeNode) .**__ne__** (*expr*)
 True when tree node does not equal *expr*. Otherwise false.
 Returns boolean.

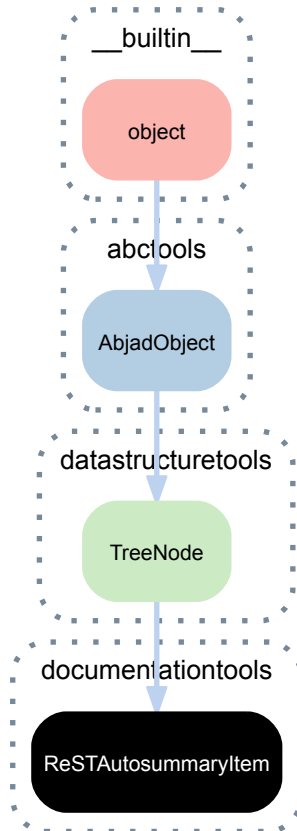
(AbjadObject) .**__repr__** ()
 Gets interpreter representation of Abjad object.
 Returns string.

`ReSTAutosummaryDirective.__setitem__(i, expr)`

Sets item *i* to *expr*.

Returns none.

35.2.16 documentationtools.ReSTAutosummaryItem



class `documentationtools.ReSTAutosummaryItem` (*name=None, text='foo'*)

A ReST autosummary item.

```
>>> item = documentationtools.ReSTAutosummaryItem(
...     text='abjad.tools.scoretools.Note')
>>> item
ReSTAutosummaryItem(
    text='abjad.tools.scoretools.Note'
)
```

```
>>> print item.rest_format
abjad.tools.scoretools.Note
```

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**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

ReSTAutosummaryItem **.rest_format**

ReST format of ReST autosummary item.

(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
```

Returns tree node.

Read/write properties

(TreeNode) .**name**

Named of tree node.

Returns string.

ReSTAutosummaryItem.**text**

Text of ReST autosummary item.

Special methods

(TreeNode) .**__copy__**(*args)

Copies tree node.

Returns new tree node.

(TreeNode) .**__deepcopy__**(*args)

Copies tree node.

Returns new tree node.

(TreeNode) .**__eq__**(expr)

True when *expr* is a tree node. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TreeNode) .**__ne__**(expr)

True when tree node does not equal *expr*. Otherwise false.

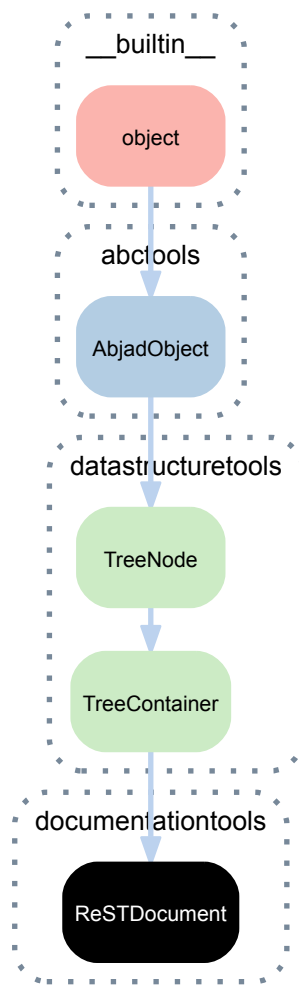
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

35.2.17 documentationtools.ReSTDocument



class `documentationtools.ReSTDocument` (*children=None, name=None*)
 A 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

```

Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(TreeContainer).children`

Children of tree container.

```

>>> 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 tree nodes.

`(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**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeContainer) **.leaves**

Leaves of tree container.

```
>>> 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.

ReSTDocument **.node_class**

Node class of ReST document.

(TreeContainer) **.nodes**

The collection of tree nodes produced by iterating tree container 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**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

ReSTDDocument **.rest_format**

ReST format of ReST document.

(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
```

Returns tree node.

Read/write properties

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

(TreeContainer) . **append** (*node*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) . **extend** (*expr*)

Extendes *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) . **index** (*node*)

Indexes *node* in tree 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 tree 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*)

Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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)
Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(*args)
Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*=’')

Formats object.

Set *format_specification* to ‘’ or ‘storage’. Interprets ‘’ equal to ‘storage’.

Returns string.

(TreeContainer).**__getitem__**(*i*)

Gets node *i* in tree 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.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode) .**__ne__**(*expr*)

True when tree node does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TreeContainer) .**__setitem__**(*i*, *expr*)

Sets *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
```

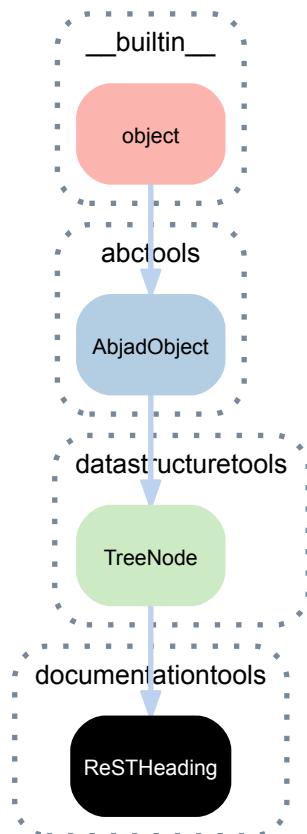
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

35.2.18 documentationtools.ReSTHeading



class `documentationtools.ReSTHeading` (*level=0, name=None, text='foo'*)
 A ReST heading.

```
>>> heading = documentationtools.ReSTHeading(
...     level=2, text='Section A')
>>> heading
ReSTHeading(
    level=2,
    text='Section A'
)
```

```
>>> print heading.rest_format
Section A
=====
```

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**

Graph order of tree node.

Returns tuple.

ReSTHeading **.heading_characters**

Heading characters of ReST heading.

(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 tree nodes.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

ReSTHeading **.rest_format**

ReST format of ReST heading.

Returns string.

(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
```

Returns tree node.

Read/write properties

`ReSTHeading.level`

Gets and sets level of ReST heading.

`(TreeNode).name`

Named of tree node.

Returns string.

`ReSTHeading.text`

Gets and sets text of ReST heading.

Returns string.

Special methods

`(TreeNode).__copy__(*args)`

Copies tree node.

Returns new tree node.

`(TreeNode).__deepcopy__(*args)`

Copies tree node.

Returns new tree node.

`(TreeNode).__eq__(expr)`

True when *expr* is a tree node. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TreeNode).__ne__(expr)`

True when tree node does not equal *expr*. Otherwise false.

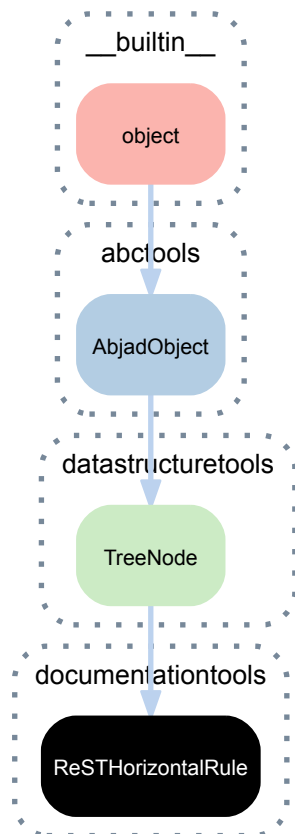
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

35.2.19 documentationtools.ReSTHorizontalRule



class `documentationtools.ReSTHorizontalRule` (*name=None*)
 A ReST horizontal rule.

```
>>> rule = documentationtools.ReSTHorizontalRule()
>>> rule
ReSTHorizontalRule()
```

```
>>> print rule.rest_format
-----
```

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**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeNode) .parent

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

ReSTHorizontalRule .rest_format

ReST format of ReSt horizontal rule.

Returns text.

(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
```

Returns tree node.

Read/write properties

(TreeNode) . **name**
Named of tree node.
Returns string.

Special methods

(TreeNode) . **__copy__** (*args)
Copies tree node.
Returns new tree node.

(TreeNode) . **__deepcopy__** (*args)
Copies tree node.
Returns new tree node.

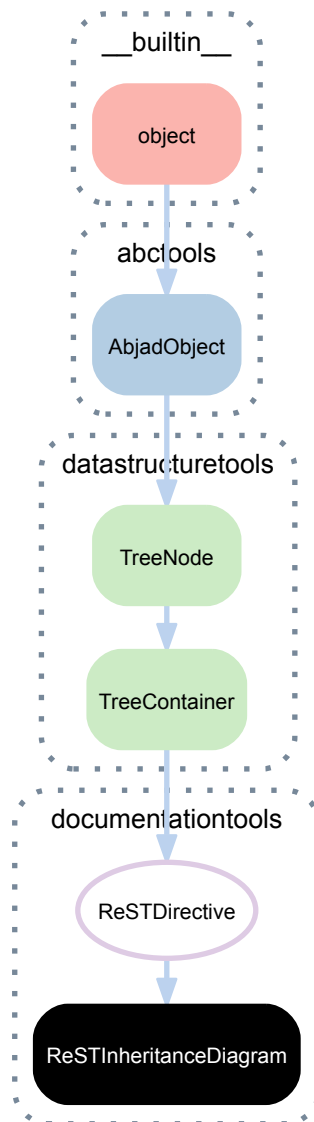
(TreeNode) . **__eq__** (expr)
True when *expr* is a tree node. Otherwise false.
Returns boolean.

(AbjadObject) . **__format__** (format_specification='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(TreeNode) . **__ne__** (expr)
True when tree node does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject) . **__repr__** ()
Gets interpreter representation of Abjad object.
Returns string.

35.2.20 documentationtools.ReSTInheritanceDiagram



class `documentationtools.ReSTInheritanceDiagram` (*argument=None, children=None, name=None, options=None*)

A ReST inheritance diagram directive.

```

>>> documentationtools.ReSTInheritanceDiagram(
...     argument=spannertools.Beam)
ReSTInheritanceDiagram(
    argument='abjad.tools.spannertools.Beam.Beam',
    options={
        'private-bases': True,
    }
)

>>> print _rest_format
.. inheritance-diagram:: abjad.tools.spannertools.Beam.Beam
:private-bases:
  
```

Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(TreeContainer) **.children**

Children of tree container.

```
>>> 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 tree nodes.

(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`
Directive of ReSt inheritance diagram.

Returns 'inheritance-diagram'.

`(TreeNode).graph_order`
Graph order of tree node.

Returns tuple.

`(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 tree nodes.

`(TreeContainer).leaves`
Leaves of tree container.

```

>>> 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**

Node class of ReST directive.

(TreeContainer) **.nodes**

The collection of tree nodes produced by iterating tree container 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**

Options of ReST directive.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(ReSTDirective) **.rest_format**
ReST format of ReST directive.

(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
```

Returns tree node.

Read/write properties

(ReSTDirective) **.argument**
Gets and sets argument of ReST directive.

(TreeNode) **.name**
Named of tree node.
Returns string.

Methods

(TreeContainer) **.append(*node*)**
Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) **.extend** (*expr*)

Extends *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)

Indexes *node* in tree 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 tree 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*)
Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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*)
Copies tree node.

Returns new tree node.

(TreeNode) .**__deepcopy__** (**args*)
Copies tree node.

Returns new tree node.

(TreeContainer) .**__delitem__** (*i*)
Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.
Returns boolean.

(AbjadObject) .**__format__** (*format_specification*='')
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(TreeContainer) .**__getitem__** (*i*)
Gets node *i* in tree 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.

(TreeContainer).**__iter__**()
Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()
Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)
True when tree node does not equal *expr*. Otherwise false.
Returns boolean.

(AbjadObject).**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(TreeContainer).**__setitem__**(*i*, *expr*)
Sets *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
```

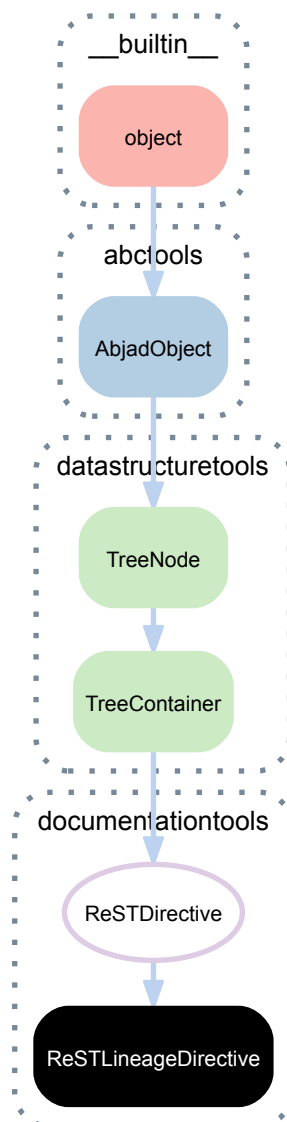
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

35.2.21 documentationtools.ReSTLineageDirective



class `documentationtools.ReSTLineageDirective` (*argument=None, children=None, name=None, options=None*)

A ReST lineage directive.

Digrams inheritance of Abjad classes.

```
>>> documentationtools.ReSTLineageDirective(argument=spannertools.Beam)
ReSTLineageDirective(
  argument='abjad.tools.spannertools.Beam.Beam'
)
```

```
>>> print _.rest_format
.. abjad-lineage:: abjad.tools.spannertools.Beam.Beam
```

Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(TreeContainer) **.children**
Children of tree container.

```
>>> 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 tree nodes.

(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**

Returns 'abjad-lineage'.

(TreeNode) **.graph_order**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeContainer) **.leaves**

Leaves of tree container.

```
>>> 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**
Node class of ReST directive.

(TreeContainer).**nodes**
The collection of tree nodes produced by iterating tree container 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**
Options of ReST directive.

(TreeNode).**parent**
Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(ReSTDirective) **.rest_format**

ReST format of ReST directive.

(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
```

Returns tree node.

Read/write properties

(ReSTDirective) **.argument**

Gets and sets argument of ReST directive.

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

(TreeContainer) **.append** (*node*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) **.extend** (*expr*)
 Extends *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)
 Indexes *node* in tree 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 tree 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*)
 Pops node *i* from tree 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 tree 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(),
  )
)

```

Returns 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*)
 Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(**args*)
 Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
 Deletes node *i* in tree container.

```

>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.

Returns boolean.

(AbjadObject) .**__format__** (*format_specification*='')
 Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TreeContainer) .**__getitem__** (*i*)
 Gets node *i* in tree 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.

(TreeContainer) .**__iter__** ()
 Iterates tree container.

Yields children of tree container.

(TreeContainer) .**__len__** ()
 Returns nonnegative integer number of nodes in container.

(TreeNode) .**__ne__** (*expr*)
 True when tree node does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject) .**__repr__** ()
 Gets interpreter representation of Abjad object.
 Returns string.

(TreeContainer).**__setitem__**(*i, expr*)

Sets *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
```

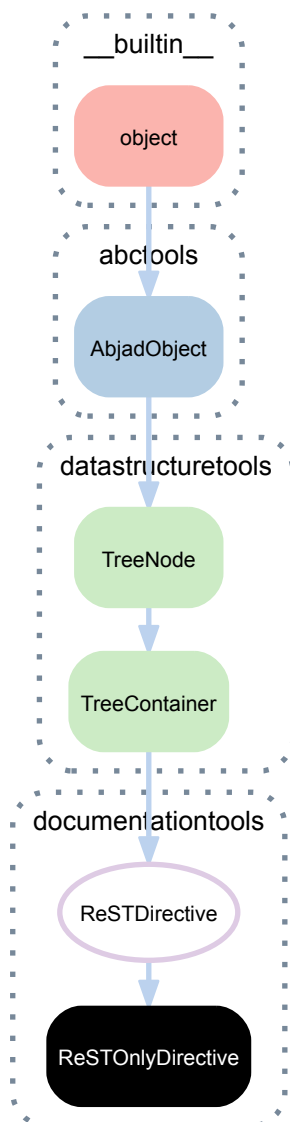
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

35.2.22 documentationtools.ReSTOnlyDirective



class `documentationtools.ReSTOnlyDirective` (*argument=None*, *children=None*,
name=None)
 A 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
```


Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`(TreeContainer).children`

Children of tree container.

```
>>> 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 tree nodes.

`(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`

Returns 'only'.

`(TreeNode).graph_order`

Graph order of tree node.

Returns tuple.

`(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 tree nodes.

`(TreeContainer).leaves`

Leaves of tree container.

```
>>> 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**
Node class of ReST directive.

(TreeContainer) **.nodes**
The collection of tree nodes produced by iterating tree container 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**
Options of ReST directive.

(TreeNode) **.parent**
Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(ReSTDirective) **.rest_format**

ReST format of ReST directive.

(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
```

Returns tree node.

Read/write properties

(ReSTDirective) **.argument**

Gets and sets argument of ReST directive.

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

(TreeContainer) **.append** (*node*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) **.extend** (*expr*)
 Extends *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer (
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)
 Indexes *node* in tree 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 tree 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*)
 Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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)
Copies tree node.

Returns new tree node.

(TreeNode).**__deepcopy__**(*args)
Copies tree node.

Returns new tree node.

(TreeContainer).**__delitem__**(*i*)
Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')
Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(TreeContainer).**__getitem__**(*i*)

Gets node *i* in tree 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.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

True when tree node does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(TreeContainer).**__setitem__**(*i*, *expr*)

Sets *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
```

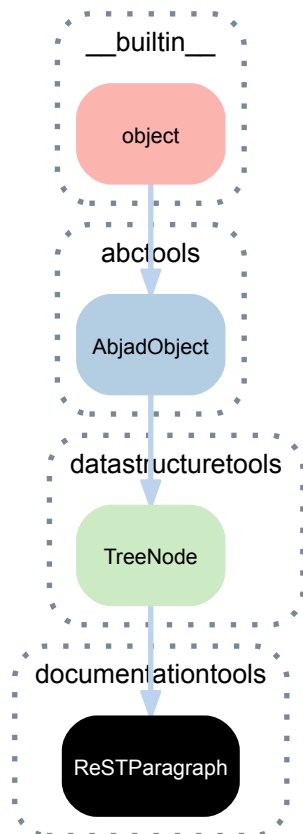
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

35.2.23 documentationtools.ReSTParagraph



class `documentationtools.ReSTParagraph` (*name=None, text='foo', wrap=True*)
A 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.

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**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

`ReSTParagraph.rest_format`

ReST format of ReST paragraph.

Returns string.

`(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
```

Returns tree node.

Read/write properties

`(TreeNode).name`

Named of tree node.

Returns string.

`ReSTParagraph.text`

Gets and sets text of ReST paragraph.

Returns string.

`ReSTParagraph.wrap`

Gets and sets wrap flag of ReST paragraph.

Returns boolean.

Special methods

`(TreeNode).__copy__(*args)`

Copies tree node.

Returns new tree node.

`(TreeNode).__deepcopy__(*args)`

Copies tree node.

Returns new tree node.

`(TreeNode).__eq__(expr)`

True when *expr* is a tree node. Otherwise false.

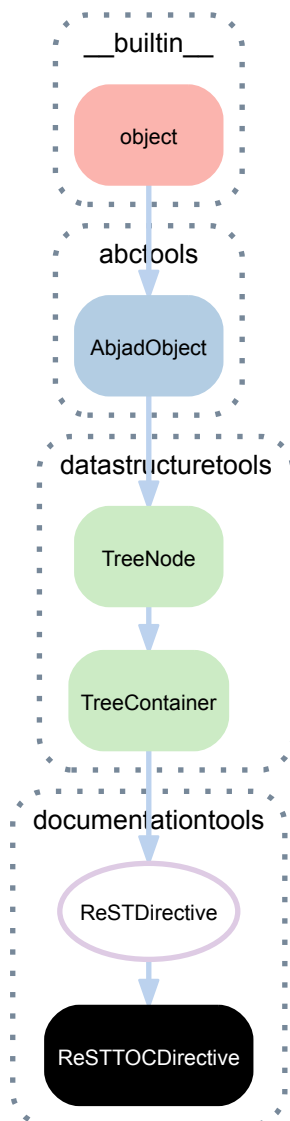
Returns boolean.

(AbjadObject) .**__format__** (*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(TreeNode) .**__ne__** (*expr*)
 True when tree node does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject) .**__repr__** ()
 Gets interpreter representation of Abjad object.
 Returns string.

35.2.24 documentationtools.ReSTTOCDirective



class documentationtools.**ReSTTOCDirective** (*argument=None, children=None, name=None, options=None*)
 A ReST TOC 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

```

Bases

- `documentationtools.ReSTDirective`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

(TreeContainer).**children**

Children of tree container.

```

>>> 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 tree nodes.

(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

Returns 'toctree'.

(TreeNode) .graph_order

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeContainer) **.leaves**

Leaves of tree container.

```
>>> 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**

Node class of ReST TOC directive.

(TreeContainer) **.nodes**

The collection of tree nodes produced by iterating tree container 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**

Options of ReST directive.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

(ReSTDirective) **.rest_format**

ReST format of ReST directive.

(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
```

Returns tree node.

Read/write properties

(ReSTDirective) **.argument**

Gets and sets argument of ReST directive.

(TreeNode) **.name**

Named of tree node.

Returns string.

Methods

(TreeContainer) **.append** (*node*)

Appends *node* to tree 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(),
  )
)
```

Returns none.

(TreeContainer) **.extend** (*expr*)

Extendes *expr* against tree container.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
  children=(
    TreeNode(),
    TreeNode(),
  )
)
```

Returns none.

(TreeContainer) **.index** (*node*)

Indexes *node* in tree 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 tree 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*)

Pops node *i* from tree 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 tree 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(),
  )
)
```

Returns 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*)
 Copies tree node.

Returns new tree node.

(TreeNode) .**__deepcopy__** (**args*)
 Copies tree node.

Returns new tree node.

(TreeContainer) .**__delitem__** (*i*)
 Deletes node *i* in tree container.

```
>>> 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 *expr* is a tree container with type, duration and children equal to this tree container. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*=‘')

Formats object.

Set *format_specification* to ‘’ or ‘*storage*’. Interprets ‘’ equal to ‘*storage*’.

Returns string.

(TreeContainer).**__getitem__**(*i*)

Gets node *i* in tree 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.

`(TreeContainer).__iter__()`
 Iterates tree container.
 Yields children of tree container.

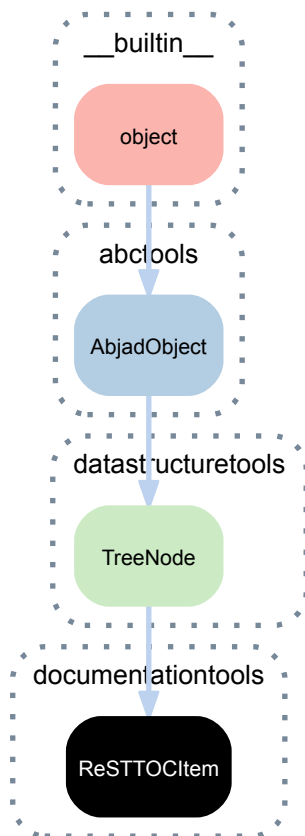
`(TreeContainer).__len__()`
 Returns nonnegative integer number of nodes in container.

`(TreeNode).__ne__(expr)`
 True when tree node does not equal *expr*. Otherwise false.
 Returns boolean.

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

`ReSTTOCDirective.__setitem__(i, expr)`
 Sets *i* to *expr*.
 Returns none.

35.2.25 documentationtools.ReSTTOCItem



class `documentationtools.ReSTTOCItem` (*name=None, text='foo'*)
 A ReST TOC item.

```

>>> item = documentationtools.ReSTTOCItem(text='api/index')
>>> item
ReSTTOCItem(
    text='api/index'
)
    
```

```
>>> print item.rest_format
api/index
```

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**

Graph order of tree node.

Returns tuple.

(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 tree nodes.

(TreeNode) **.parent**

Parent of tree 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
```

Returns tree node.

(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 tree nodes.

ReSTTOCItem.**rest_format**

ReST format of ReST TOC item.

Returns string.

(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
```

Returns tree node.

Read/write properties

(TreeNode) **.name**

Named of tree node.

Returns string.

ReSTTOCItem.**text**

Gets and sets text of ReST TOC item.

Returns string.

Special methods

(TreeNode) **.__copy__** (*args)

Copies tree node.

Returns new tree node.

(TreeNode) **.__deepcopy__** (*args)

Copies tree node.

Returns new tree node.

(TreeNode) **.__eq__** (expr)

True when *expr* is a tree node. Otherwise false.

Returns boolean.

(AbjadObject) **.__format__** (format_specification='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

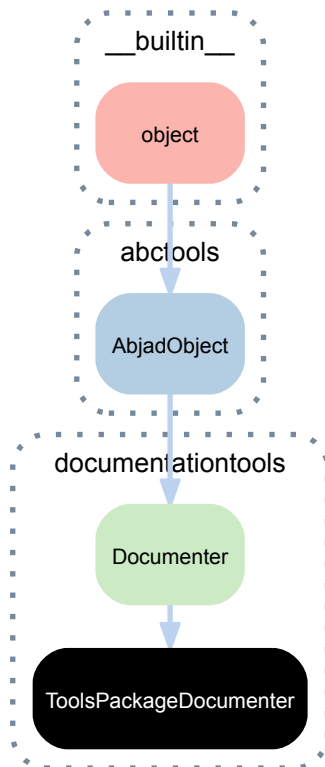
(TreeNode) **.__ne__** (expr)

True when tree node does not equal *expr*. Otherwise false.

Returns boolean.

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

35.2.26 documentationtools.ToolsPackageDocumenter



class `documentationtools.ToolsPackageDocumenter` (*object_=None,* *nored_directory_names=(),* *ig-*
fix='abjad.tools.' *pre-*
 Generates an index for every tools package.

Bases

- `documentationtools.Documenter`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ToolsPackageDocumenter.abstract_class_documenters`
 Abstract class documenters.

`ToolsPackageDocumenter.all_documenters`
 All documenters.

`ToolsPackageDocumenter.concrete_class_documenters`
 Concrete class documenters.

`ToolsPackageDocumenter.documentation_section`
 Documentation section.

`ToolsPackageDocumenter.function_documenters`
Function documenters.

`ToolsPackageDocumenter.ignored_directory_names`
Ignored directory names.

`ToolsPackageDocumenter.module_name`
Module name.

Returns string.

`(Documenter).object_`
Object of documenter.

`(Documenter).prefix`
Prefix of documenter.

Methods

`ToolsPackageDocumenter.create_api_toc_section()`
Creates a TOC section to be included in the master API index.

```
>>> module = scoretools
>>> documenter = documentationtools.ToolsPackageDocumenter(module)
>>> result = documenter.create_api_toc_section()
```

Returns list.

Static methods

`(Documenter).write(file_path, restructured_text)`
Writes *restructured_text* to *file_path*.

Returns none.

Special methods

`ToolsPackageDocumenter.__call__()`
Calls tools package documenter.

Generates documentation:

```
>>> module = scoretools
>>> documenter = documentationtools.ToolsPackageDocumenter(
...     scoretools)
>>> restructured_text = documenter()
```

Returns string.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(Documenter).__makenew__(object_=None, prefix=None)`
Makes new documenter.

Returns new documenter.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

35.3 Functions

35.3.1 `documentationtools.compare_images`

`documentationtools.compare_images(image_one, image_two)`
Compare *image_one* against *image_two* using ImageMagick's *compare* commandline tool.
Returns true if images are the same. Otherwise false.
Returns false if *compare* is not available.

35.3.2 `documentationtools.list_all_abjad_classes`

`documentationtools.list_all_abjad_classes(modules=None)`
Lists all public classes defined in Abjad.

```
>>> all_classes = documentationtools.list_all_abjad_classes()
```

35.3.3 `documentationtools.list_all_abjad_functions`

`documentationtools.list_all_abjad_functions(modules=None)`
Lists all public functions defined in Abjad.

```
>>> all_functions = documentationtools.list_all_abjad_functions()
```

35.3.4 `documentationtools.make_ligeti_example_lilypond_file`

`documentationtools.make_ligeti_example_lilypond_file(music=None)`
Make Ligeti example LilyPond file.
Returns LilyPond file.

35.3.5 `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.

35.3.6 `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.

35.3.7 documentationtools.make_text_alignment_example_lilypond_file

`documentationtools.make_text_alignment_example_lilypond_file` (*music=None*)

Make text-alignment example LilyPond file with *music*.

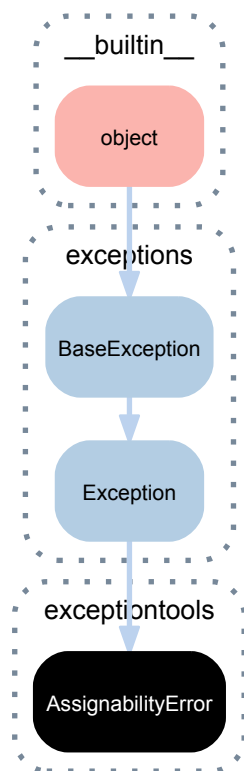
```
>>> score = Score([Staff('c d e f')])
>>> lilypond_file = documentationtools.make_text_alignment_example_lilypond_file(score)
```

Returns LilyPond file.

EXCEPTIONTOOLS

36.1 Concrete classes

36.1.1 `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.

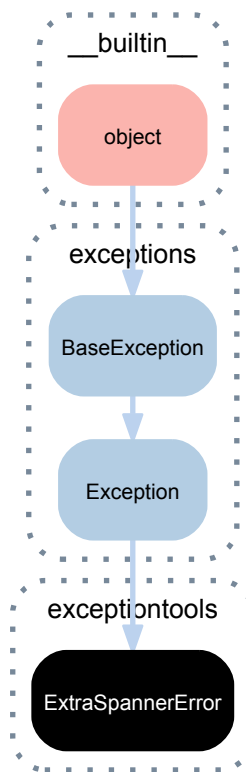
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.2 exceptiontools.ExtraSpannerError



class exceptiontools.**ExtraSpannerError**
 More than one spanner found for single-spanner operation.

Bases

- exceptiontools.Exception
- exceptiontools.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.

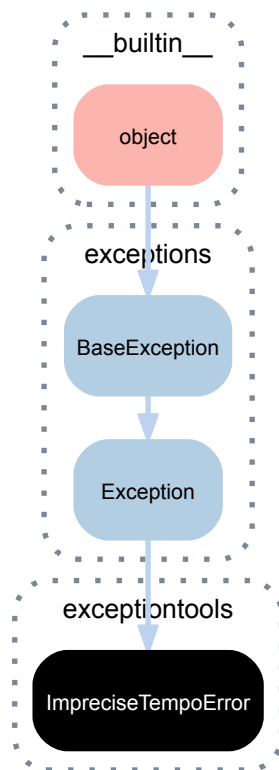
```
(BaseException).__repr__() <==> repr(x)
```

```
(BaseException).__setattr__()
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException).__str__() <==> str(x)
```

```
(BaseException).__unicode__()
```

36.1.3 exceptiontools.ImpreciseTempoError



```
class exceptiontools.ImpreciseTempoError
    Tempo 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.

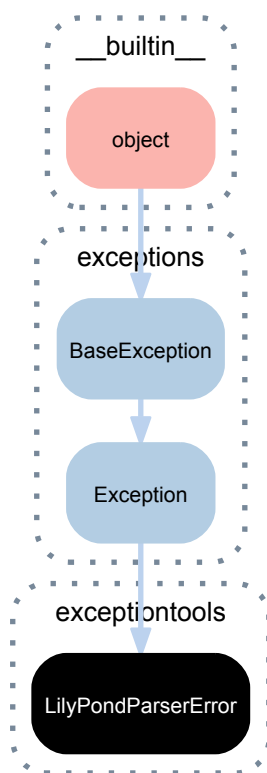
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.4 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.

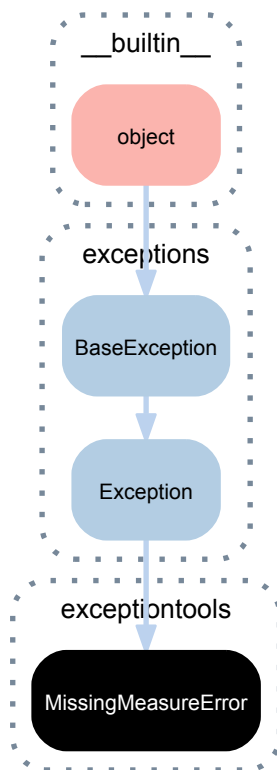
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.5 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.

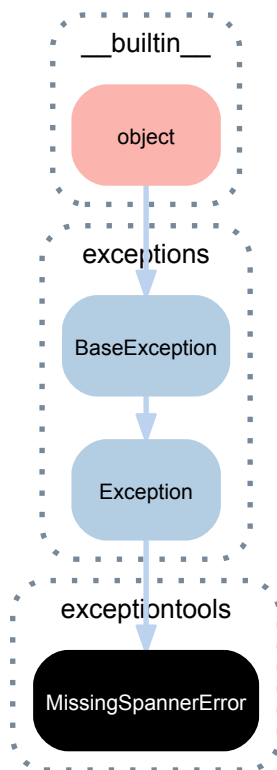
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.6 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.

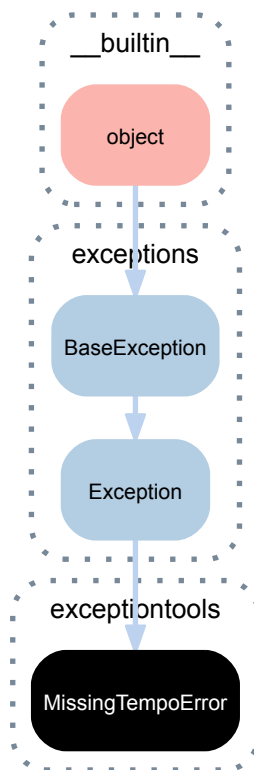
```
(BaseException).__repr__() <==> repr(x)
```

```
(BaseException).__setattr__()
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException).__str__() <==> str(x)
```

```
(BaseException).__unicode__()
```

36.1.7 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.

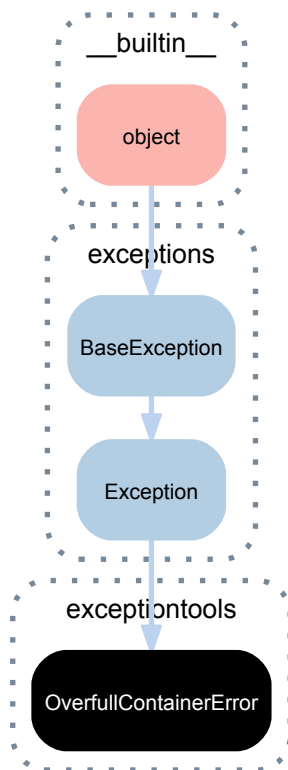
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.8 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.

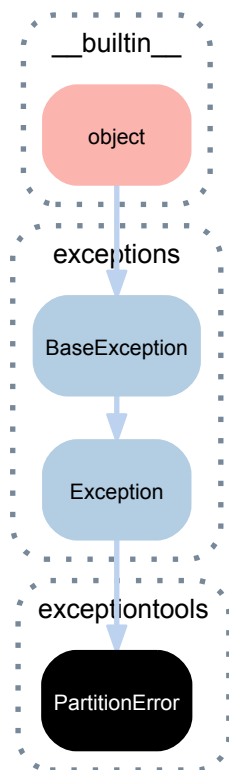
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.9 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.

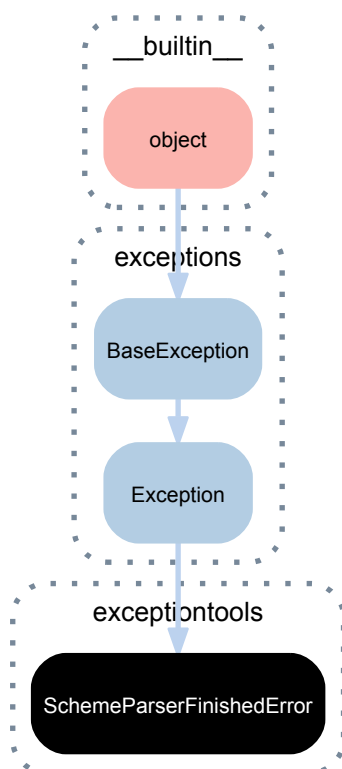
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.10 exceptiontools.SchemeParserFinishedError



```
class exceptiontools.SchemeParserFinishedError
    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.

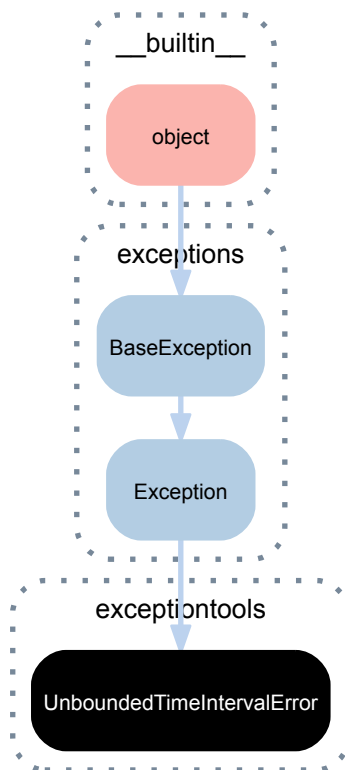
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.11 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.

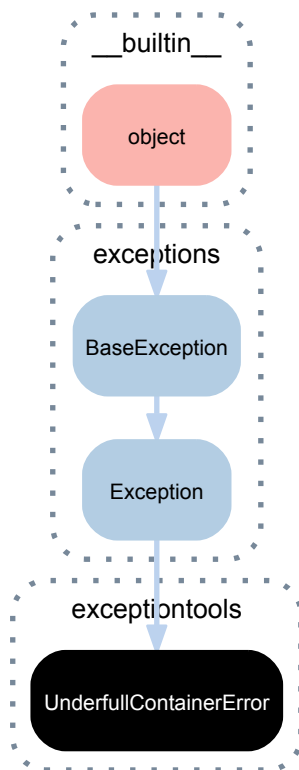
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

36.1.12 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)
```

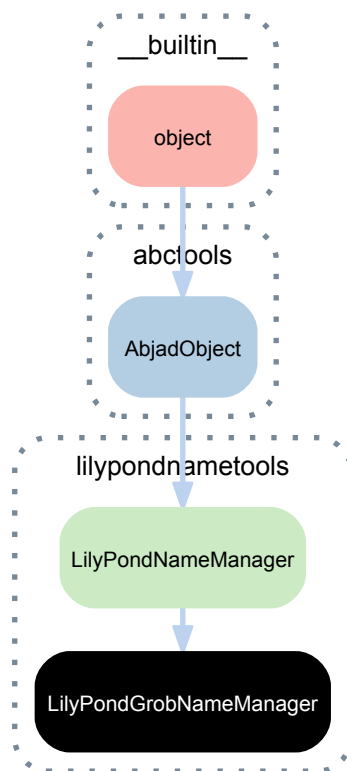
```
(BaseException) .__setattr__()  
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException) .__str__() <==> str(x)
```

```
(BaseException) .__unicode__()
```


37.1 Concrete classes

37.1.1 lilypondnametools.LilyPondGrobNameManager



class `lilypondnametools.LilyPondGrobNameManager`
LilyPond grob name manager.

Bases

- `lilypondnametools.LilyPondNameManager`
- `abctools.AbjadObject`
- `__builtin__.object`

Special methods

`(LilyPondNameManager).__eq__(arg)`

True when *arg* is a LilyPond name manager with attribute pairs equal to those of this LilyPond name

manager. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

LilyPondGrobNameManager.**__getattr__**(*name*)

Gets attribute *name* from LilyPond grob name manager.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(LilyPondNameManager).**__repr__**()

Gets interpreter representation of LilyPond name manager.

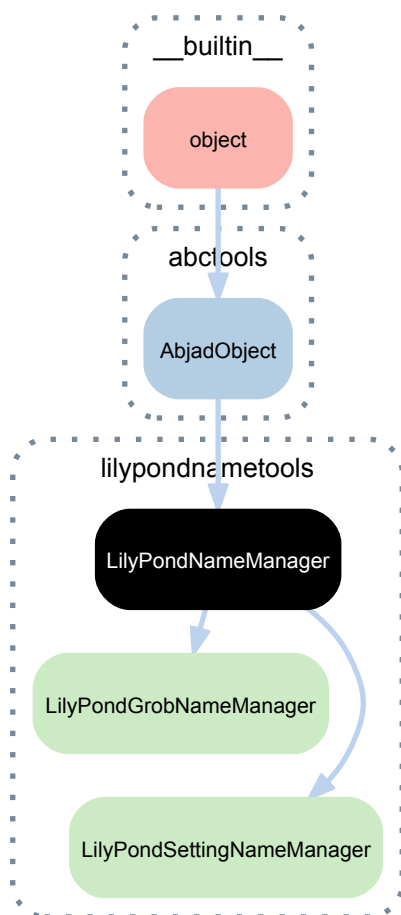
Returns string.

LilyPondGrobNameManager.**__setattr__**(*attribute_name*, *value*)

Sets attribute *attribute_name* of grob name manager to *value*.

Returns none.

37.1.2 lilypondnametools.LilyPondNameManager



class `lilypondnametools.LilyPondNameManager`

Base class from which LilyPond grob and setting managers inherit.

Bases

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

Special methods

`LilyPondNameManager.__eq__(arg)`

True when *arg* is a LilyPond name manager with attribute pairs equal to those of this LilyPond name manager. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

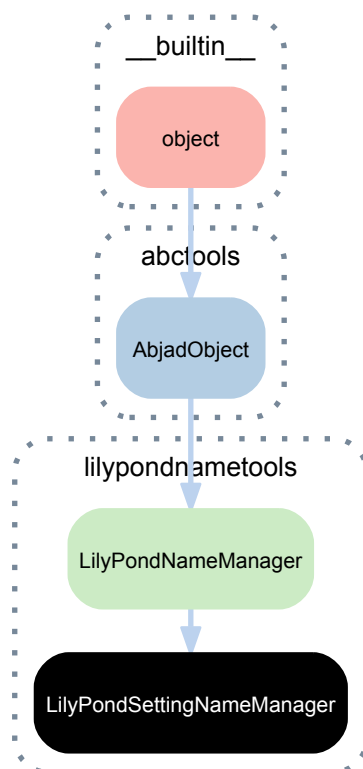
Returns boolean.

`LilyPondNameManager.__repr__()`

Gets interpreter representation of LilyPond name manager.

Returns string.

37.1.3 lilypondnametools.LilyPondSettingNameManager



class `lilypondnametools.LilyPondSettingNameManager`
LilyPond setting name manager.

Bases

- `lilypondnametools.LilyPondNameManager`
- `abctools.AbjadObject`
- `__builtin__.object`

Special methods

`(LilyPondNameManager).__eq__(arg)`

True when *arg* is a LilyPond name manager with attribute pairs equal to those of this LilyPond name manager. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`LilyPondSettingNameManager.__getattr__(name)`

Gets setting *name* from LilyPond setting name manager.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

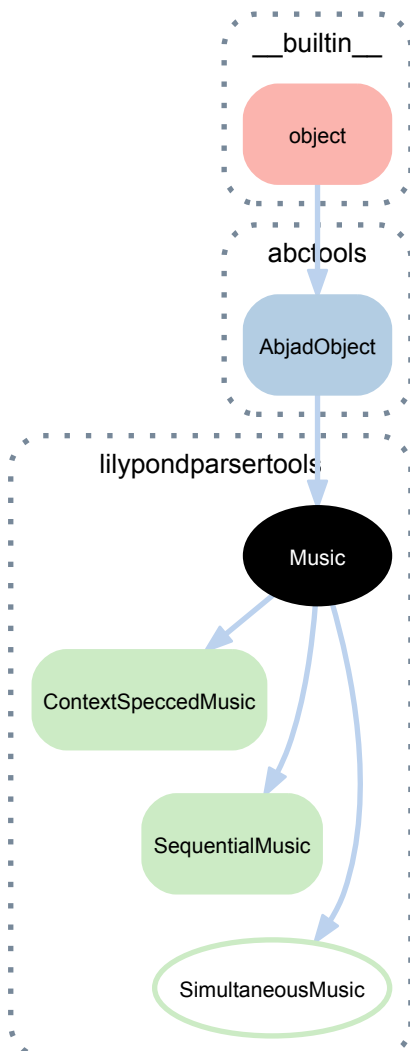
`(LilyPondNameManager).__repr__()`

Gets interpreter representation of LilyPond name manager.

Returns string.

38.1 Abstract classes

38.1.1 lilypondparsertools.Music



class lilypondparsertools.**Music** (*music=None*)
Abjad model of the LilyPond AST music node.

Bases

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

Methods

`Music.construct()`
Please document.

Special methods

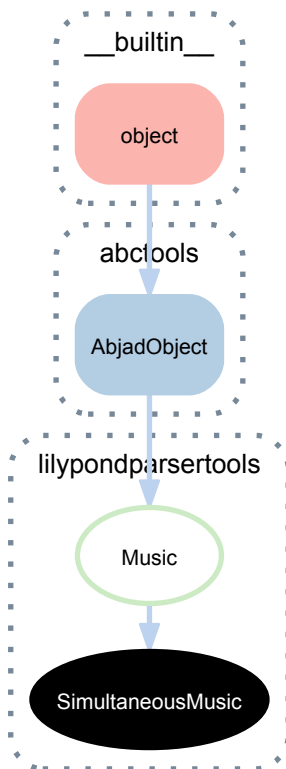
`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

38.1.2 lilypondparsertools.SimultaneousMusic



class lilypondparsertools.**SimultaneousMusic** (*music=None*)
 Abjad model of the LilyPond AST simultaneous music node.

Bases

- lilypondparsertools.Music
- abctools.AbjadObject
- __builtin__.object

Methods

(Music).**construct**()
 Please document.

Special methods

(AbjadObject).**__eq__**(*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(AbjadObject).**__format__**(*format_specification=''*)
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

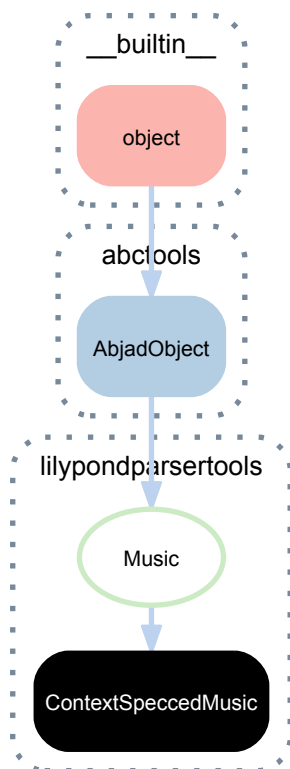
(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

38.2 Concrete classes

38.2.1 lilypondparsertools.ContextSpeccedMusic



```
class lilypondparsertools.ContextSpeccedMusic (context_name=None, optional_id=None,  
                                              optional_context_mod=None,      mu-  
                                              sic=None)
```

Abjad model of the LilyPond AST context-specced music node.

Bases

- `lilypondparsertools.Music`
- `abctools.AbjadObject`
- `__builtin__.object`

Read-only properties

`ContextSpeccedMusic.known_contexts`

Known contexts.

Returns dictionary.

Methods

`ContextSpeccedMusic.construct()`

Constructs context.

Returns context.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

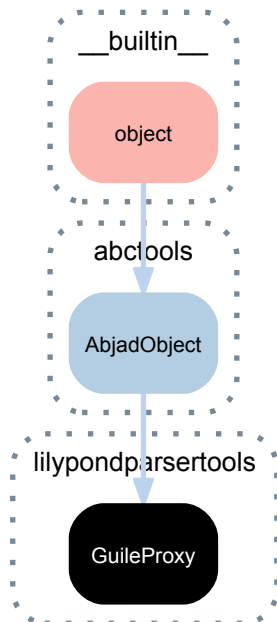
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

38.2.2 lilypondparsertools.GuileProxy



class `lilypondparsertools.GuileProxy(client=None)`

Emulates LilyPond music functions.

Used internally by LilyPondParser.

Not composer-safe.

Bases

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

Methods

`GuileProxy.acciaccatura` (*music*)
Handles LilyPond \acciaccatura command.

`GuileProxy.appoggiatura` (*music*)
Handles LilyPond \appoggiatura command.

`GuileProxy.bar` (*string*)
Handles LilyPond \bar command.

`GuileProxy.breathe` ()
Handles LilyPond \breathe command.

`GuileProxy.clef` (*string*)
Handles LilyPond \clef command.

`GuileProxy.grace` (*music*)
Handles LilyPond \grace command.

`GuileProxy.key` (*notename_pitch, number_list*)
Handles LilyPond \key command.

`GuileProxy.language` (*string*)
Handles LilyPond \language command.

`GuileProxy.makeClusters` (*music*)
Handles LilyPond \makeClusters command.

`GuileProxy.mark` (*label*)
Handles LilyPond \mark command.

`GuileProxy.one_voice` ()
Handles LilyPond \oneVoice command.

`GuileProxy.relative` (*pitch, music*)
Handles LilyPond \relative command.

`GuileProxy.skip` (*duration*)
Handles LilyPond \skip command.

`GuileProxy.slashed_grace_container` (*music*)
Handles LilyPond \slahsedGrace command.

`GuileProxy.time` (*number_list, fraction*)
Handles LilyPond \time command.

`GuileProxy.times` (*fraction, music*)
Handles LilyPond \times command.

`GuileProxy.transpose` (*from_pitch, to_pitch, music*)
Handles LilyPond \transpose command.

`GuileProxy.voiceFour` ()
Handles LilyPond \voiceFour command.

`GuileProxy.voiceOne` ()
Handles LilyPond \voiceOnce command.

`GuileProxy.voiceThree` ()
Handles LilyPond \voiceThree command.

`GuileProxy.voiceTwo()`
 Handles LilyPond `\voiceTwo` command.

Special methods

`GuileProxy.__call__(function_name, args)`
 Calls Guile proxy on *function_name* with *args*.
 Returns function output.

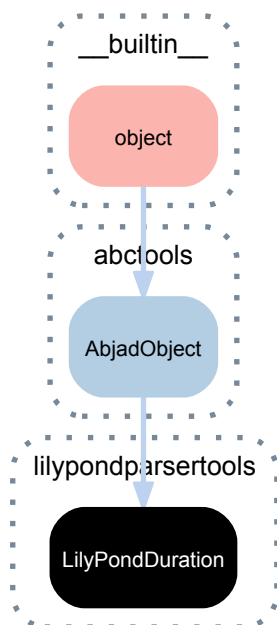
`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

38.2.3 lilypondparsertools.LilyPondDuration



class `lilypondparsertools.LilyPondDuration` (*duration=None, 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)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

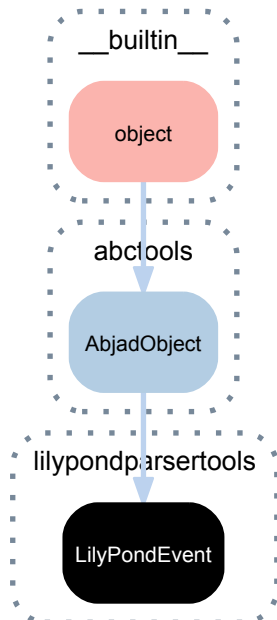
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

38.2.4 lilypondparsertools.LilyPondEvent



class `lilypondparsertools.LilyPondEvent` (*name=None, **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)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

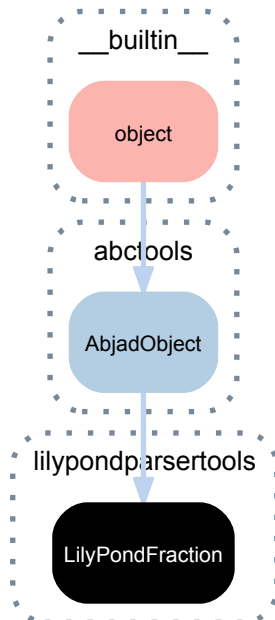
Returns boolean.

`LilyPondEvent.__repr__()`

Gets interpreter representation of LilyPond event.

Returns string.

38.2.5 lilypondparsertools.LilyPondFraction



class `lilypondparsertools.LilyPondFraction` (*numerator=0, denominator=1*)

Model of a fraction in LilyPond.

Not composer-safe.

Used internally by LilyPondParser.

Bases

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

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

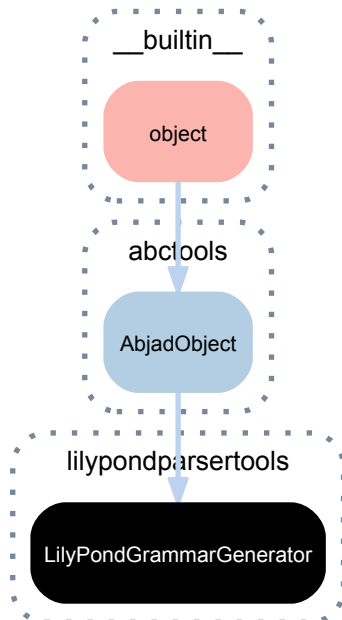
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

38.2.6 lilypondparsertools.LilyPondGrammarGenerator



class `lilypondparsertools.LilyPondGrammarGenerator`
Generates 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*)
 Calls LilyPond grammar generator.

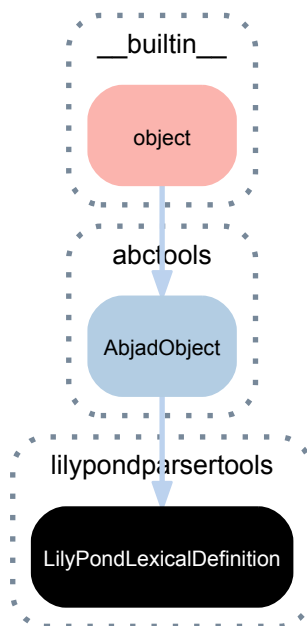
`(AbjadObject).__eq__` (*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__` (*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

`(AbjadObject).__ne__` (*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

38.2.7 lilypondparsertools.LilyPondLexicalDefinition



class `lilypondparsertools.LilyPondLexicalDefinition` (*client=None*)
 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*)
`%[^\n][^\n]*[^\n]`

`LilyPondLexicalDefinition.t_INITIAL_markup_notes_216` (*t*)
`%[^\n]`

`LilyPondLexicalDefinition.t_INITIAL_markup_notes_218` (*t*)
`%[^\n]`

`LilyPondLexicalDefinition.t_INITIAL_markup_notes_220` (*t*)
`%[^\n][^\n]*`

`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[ntfr]*`

`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)
  [^\{ } "\ tnr]+
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

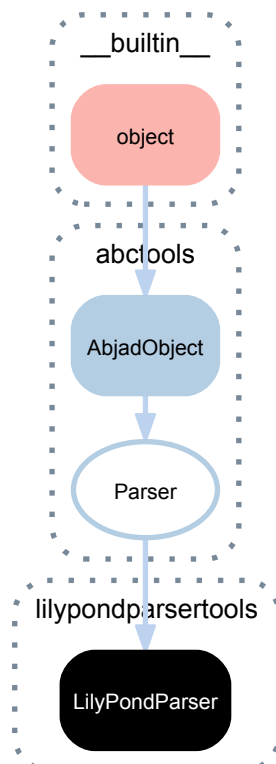
(AbjadObject).**__eq__**(*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

38.2.8 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)
>>> print format(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)
>>> print format(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`

Lexer rules object of LilyPond parser.

`(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.

`LilyPondParser.parser_rules_object`

Parser rules object of LilyPond parser.

`(Parser).pickle_path`

The output path for the parser's pickled parsing tables.

Read/write properties

`LilyPondParser.default_language`

Gets and sets default language of parser.

```
>>> 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)`

Registers 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()`

Lists all LilyPond contexts recognized by LilyPond parser.

```
>>> 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
PetrucchiStaff
PetrucchiVoice
PianoStaff
RhythmicStaff
Score
Staff
StaffGroup
TabStaff
TabVoice
VaticanaStaff
VaticanaVoice
Voice
```

Returns list.

`LilyPondParser.list_known_grobs()`

Lists all LilyPond grobs recognized by LilyPond parser.

```
>>> 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
StaffGroupier
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()`

Lists all markup functions recognized by LilyPond parser.

```
>>> 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()`

Lists all music functions recognized by LilyPond parser.

```
>>> 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)`

Calls LilyPond parser on *input_string*.

Returns Abjad components.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

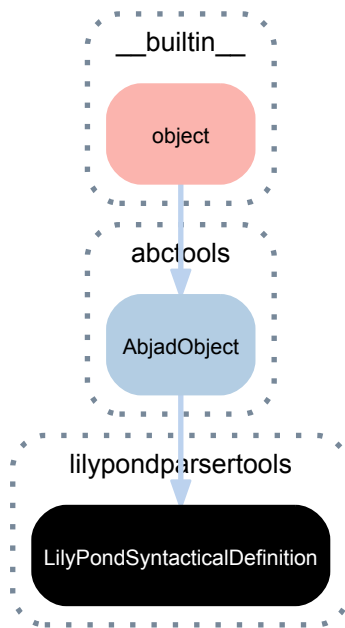
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

38.2.9 lilypondparsertools.LilyPondSyntacticalDefinition



class `lilypondparsertools.LilyPondSyntacticalDefinition` (*client=None*)
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_composite_music__music_bare** (*p*)
 composite_music : music_bare

LilyPondSyntacticalDefinition.**p_context_change__CHANGE__STRING__Chr61__STRING** (*p*)
 context_change : CHANGE STRING '=' STRING

LilyPondSyntacticalDefinition.**p_context_def_spec_block__CONTEXT__Chr123__context_def_spec_block**
 context_def_spec_block : CONTEXT '{ 'context_def_spec_body '}'

LilyPondSyntacticalDefinition.**p_context_def_spec_body__CONTEXT_DEF_IDENTIFIER** (*p*)
 context_def_spec_body : CONTEXT_DEF_IDENTIFIER

```

LilyPondSyntacticalDefinition.p_context_def_spec_body__Empty (p)
    context_def_spec_body :
LilyPondSyntacticalDefinition.p_context_def_spec_body__context_def_spec_body__context_mod
    context_def_spec_body : context_def_spec_body context_mod
LilyPondSyntacticalDefinition.p_context_def_spec_body__context_def_spec_body__context_modification
    context_def_spec_body : context_def_spec_body context_modification
LilyPondSyntacticalDefinition.p_context_def_spec_body__context_def_spec_body__embedded_scm
    context_def_spec_body : context_def_spec_body embedded_scm
LilyPondSyntacticalDefinition.p_context_mod__property_operation (p)
    context_mod : property_operation
LilyPondSyntacticalDefinition.p_context_mod_list__Empty (p)
    context_mod_list :
LilyPondSyntacticalDefinition.p_context_mod_list__context_mod_list__CONTEXT_MOD_IDENTIFIER
    context_mod_list : context_mod_list CONTEXT_MOD_IDENTIFIER
LilyPondSyntacticalDefinition.p_context_mod_list__context_mod_list__context_mod (p)
    context_mod_list : context_mod_list context_mod
LilyPondSyntacticalDefinition.p_context_mod_list__context_mod_list__embedded_scm (p)
    context_mod_list : context_mod_list embedded_scm
LilyPondSyntacticalDefinition.p_context_modification__CONTEXT_MOD_IDENTIFIER (p)
    context_modification : CONTEXT_MOD_IDENTIFIER
LilyPondSyntacticalDefinition.p_context_modification__WITH__CONTEXT_MOD_IDENTIFIER (p)
    context_modification : WITH CONTEXT_MOD_IDENTIFIER
LilyPondSyntacticalDefinition.p_context_modification__WITH__Chr123__context_mod_list__CH
    context_modification : WITH '{ context_mod_list '}'
LilyPondSyntacticalDefinition.p_context_modification__WITH__embedded_scm_closed (p)
    context_modification : WITH embedded_scm_closed
LilyPondSyntacticalDefinition.p_context_prop_spec__simple_string (p)
    context_prop_spec : simple_string
LilyPondSyntacticalDefinition.p_context_prop_spec__simple_string__Chr46__simple_string (p)
    context_prop_spec : simple_string '.' simple_string
LilyPondSyntacticalDefinition.p_direction_less_char__Chr126 (p)
    direction_less_char : '~'
LilyPondSyntacticalDefinition.p_direction_less_char__Chr40 (p)
    direction_less_char : '('
LilyPondSyntacticalDefinition.p_direction_less_char__Chr41 (p)
    direction_less_char : ')'
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 (p)
    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 (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_closed_keep duration_length

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_PITCH__function_arglist_backup : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_keep pitch_also_in_chords (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_keep pitch_also_in_chords

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_backup BACKUP (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_backup BACKUP

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' NUM-BER_IDENTIFIER (p)
    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_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' REAL (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' UNSIGNED (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' UNSIGNED

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep FRAC-TION (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep FRAC-TION

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep NUM-BER_IDENTIFIER (p)
    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_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep REAL (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep REAL

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep UNSIGNED (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep UNSIGNED

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep post_event_nofinger (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep post_event_nofinger

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_keep embed-ded_scm_arg_closed (p)
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_keep embed-ded_scm_arg_closed

```

```

LilyPondSyntacticalDefinition.p_function_arglist_backup__function_arglist_backup__REPARSE
    function_arglist_backup : function_arglist_backup REPARSE bare_number

LilyPondSyntacticalDefinition.p_function_arglist_backup__function_arglist_backup__REPARSE
    function_arglist_backup : function_arglist_backup REPARSE embedded_scm_arg_closed

LilyPondSyntacticalDefinition.p_function_arglist_backup__function_arglist_backup__REPARSE
    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_co
    function_arglist_closed_keep : function_arglist_closed_common

LilyPondSyntacticalDefinition.p_function_arglist_closed_optional__EXPECT_OPTIONAL__EXPEC
    function_arglist_closed_optional : EXPECT_OPTIONAL EXPECT_DURATION func-
    tion_arglist_closed_optional

```

```

LilyPondSyntacticalDefinition.p_function_arglist_closed_optional__EXPECT_OPTIONAL__EXPE
function_arglist_closed_optional      :      EXPECT_OPTIONAL      EXPECT_PITCH      func-
tion_arglist_closed_optional

LilyPondSyntacticalDefinition.p_function_arglist_closed_optional__function_arglist_backu
function_arglist_closed_optional : function_arglist_backup BACKUP

LilyPondSyntacticalDefinition.p_function_arglist_closed_optional__function_arglist_clos
function_arglist_closed_optional : function_arglist_closed_keep %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_function_arglist_common__EXPECT_SCM__function_arglist_cl
function_arglist_common : EXPECT_SCM function_arglist_closed_optional bare_number

LilyPondSyntacticalDefinition.p_function_arglist_common__EXPECT_SCM__function_arglist_cl
function_arglist_common : EXPECT_SCM function_arglist_closed_optional fraction

LilyPondSyntacticalDefinition.p_function_arglist_common__EXPECT_SCM__function_arglist_cl
function_arglist_common : EXPECT_SCM function_arglist_closed_optional post_event_nofinger

LilyPondSyntacticalDefinition.p_function_arglist_common__EXPECT_SCM__function_arglist_op
function_arglist_common : 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_argl
function_arglist_common_minus :      EXPECT_SCM      function_arglist_closed_optional  '-'  NUM-
BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_common_minus__EXPECT_SCM__function_argl
function_arglist_common_minus : EXPECT_SCM function_arglist_closed_optional '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_common_minus__EXPECT_SCM__function_argl
function_arglist_common_minus : EXPECT_SCM function_arglist_closed_optional '-' UNSIGNED

LilyPondSyntacticalDefinition.p_function_arglist_common_minus__function_arglist_common_m
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_DURA
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_closed dura-
tion_length

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_PITO
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  embed-
ded_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 ‘-‘ UN-SIGNED

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_list (p)
    markup_braced_list_body : markup_braced_list_body markup_list

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_command_list_arguments (p)
    markup_command_basic_arguments : EXPECT_SCM markup_command_list_arguments embed-
    ded_scm_closed

```

```

LilyPondSyntacticalDefinition.p_markup_command_list__MARKUP_LIST_FUNCTION__markup_comman
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(*p*)
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
    simple_element : pitch exclamations questions octave_check optional_notemode_duration optional_rest

LilyPondSyntacticalDefinition.p_simple_markup__MARKUP_FUNCTION__markup_command__basic_arguments
    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
    simple_music_property_def : OVERRIDE context_prop_spec property_path '=' scalar

LilyPondSyntacticalDefinition.p_simple_music_property_def__REVERT__context_prop_spec__embedded_scm
    simple_music_property_def : REVERT context_prop_spec embedded_scm

LilyPondSyntacticalDefinition.p_simple_music_property_def__SET__context_prop_spec__Chr61__scalar
    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
    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_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)
```

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

```
(AbjadObject).__format__(format_specification='')
```

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

```
(AbjadObject).__ne__(expr)
```

Is true when Abjad object does not equal *expr*. Otherwise false.

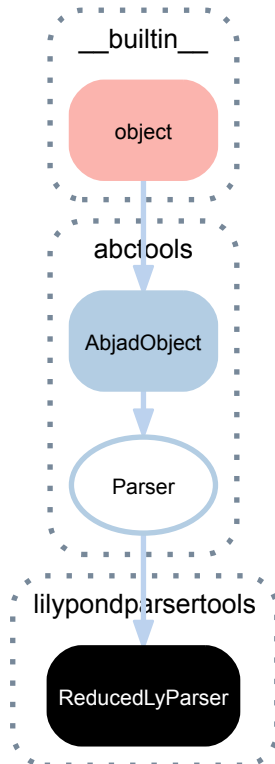
Returns boolean.

```
(AbjadObject).__repr__()
```

Gets interpreter representation of Abjad object.

Returns string.

38.2.10 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)
>>> print format(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)
>>> print format(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)
>>> print format(result)
{
    c'4
    d'4
}
```

Understands LilyPond-like default durations:

```
>>> string = "c'4 d' e' f'"
>>> result = parser(string)
>>> print format(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)
>>> print format(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)
>>> print format(result)
{
    c16 [ (
        d16 ~
        d16 )
    f16 ]
}
```

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**
Lexer rules object of reduced ly parser.
- (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 rules object of reduced ly parser.
- (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*)
 tuplet : 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](fflsslflsltqfltqslqlf)?

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.

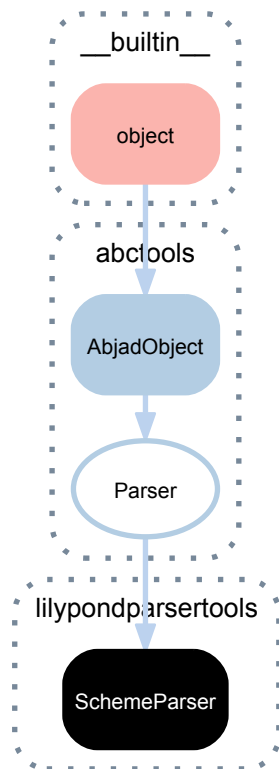
(AbjadObject) **.__eq__** (*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(AbjadObject) **.__format__** (*format_specification*='')
 Formats object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject) **.__ne__** (*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject) **.__repr__** ()
 Gets interpreter representation of Abjad object.
 Returns string.

38.2.11 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 *SchemeParserFinishedError*.

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*.

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`
 Lexer rules object of Scheme parser.

`(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 rules object of Scheme parser.

(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!\$%&*/<>?~_^:=][A-Za-z0-9!\$%&*/<>?~_^:=.+-]*|[-]|...)

`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.

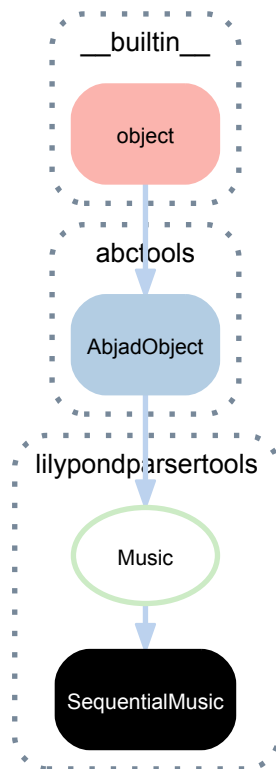
`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

38.2.12 lilypondparsertools.SequentialMusic



class `lilypondparsertools.SequentialMusic` (*music=None*)
 Abjad model of the LilyPond AST sequential music node.

Bases

- `lilypondparsertools.Music`
- `abctools.AbjadObject`
- `__builtin__.object`

Methods

`SequentialMusic.construct()`

Constructs sequential music.

Returns Abjad container.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

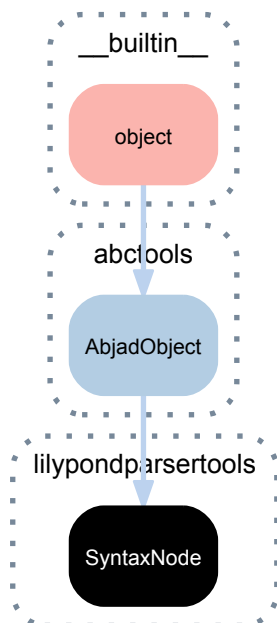
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(AbjadObject).**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

38.2.13 lilypondparsertools.SyntaxNode



class lilypondparsertools.**SyntaxNode** (*type=None, value=None*)
 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*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

(AbjadObject).**__format__**(*format_specification=''*)
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`SyntaxNode.__getitem__(item)`

Gets *item* from syntax node.

Returns item.

`SyntaxNode.__len__()`

Length of syntax node.

Returns nonnegative integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`SyntaxNode.__repr__()`

Gets interpreter representation of syntax node.

Returns string.

`SyntaxNode.__str__()`

String representation of syntax node.

Returns string.

38.3 Functions

38.3.1 lilypondparsertools.parse_reduced_ly_syntax

`lilypondparsertools.parse_reduced_ly_syntax(string)`

Parse the reduced LilyPond rhythmic syntax:

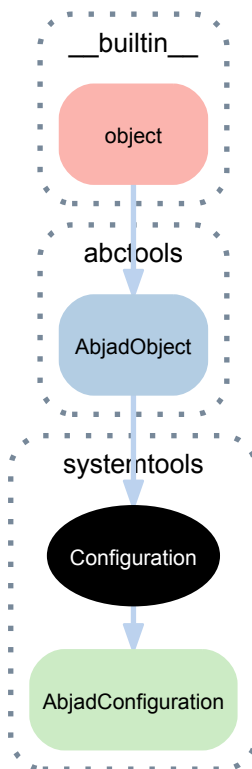
```
>>> string = '4 -4. 8.. 5/3 { } 4'
>>> result = lilypondparsertools.parse_reduced_ly_syntax(string)
```

```
>>> for x in result:
...     x
...
Note("c'4")
Rest('r4.')
Note("c'8..")
Tuplet(Multiplier(5, 3), '')
Note("c'4")
```

Returns list.

39.1 Abstract classes

39.1.1 systemtools.Configuration



class `systemtools.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)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`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)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

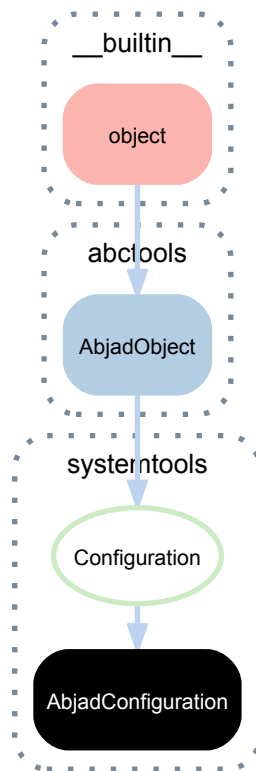
`Configuration.__setitem__(i, arg)`

Sets setting *i* to *arg*.

Returns none.

39.2 Concrete classes

39.2.1 systemtools.AbjadConfiguration



class systemtools.**AbjadConfiguration**
 Abjad configuration.

```
>>> ABJCONFIG = systemtools.AbjadConfiguration()
>>> ABJCONFIG['accidental_spelling']
'mixed'
```

```
>>> configuration = systemtools.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

- systemtools.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 2.14'
```

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_version_string()`
Gets Abjad version string.

```
>>> abjad_configuration.get_abjad_version_string()
'2.14'
```

Returns string.

`AbjadConfiguration.get_lilypond_version_string()`
Gets LilyPond version string:

```
>>> abjad_configuration.get_lilypond_version_string()
'2.17.28'
```

Returns string.

`AbjadConfiguration.get_python_version_string()`
Gets Python version string.

```
>>> abjad_configuration.get_python_version_string()
'2.7.5'
```

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', 'pytest': '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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(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*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

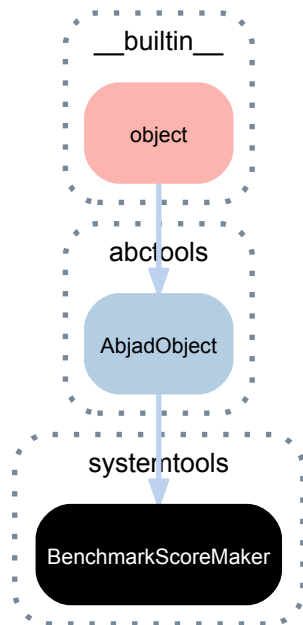
(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

(Configuration).**__setitem__**(*i*, *arg*)
 Sets setting *i* to *arg*.
 Returns none.

39.2.2 systemtools.BenchmarkScoreMaker



class systemtools.BenchmarkScoreMaker
 Benchmark score maker:

```
>>> benchmark_score_maker = systemtools.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_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_score_with_indicators_01**()

Make 200-note voice with dynamic on every 20th note:

2.12	(r9704)	initialization:	630,433	function calls
2.12	(r9710)	initialization:	235,120	function calls
2.12	(r9726)	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_score_with_indicators_02**()

Make 200-note staff with dynamic 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_score_with_indicators_03**()

Make 200-note staff with dynamic 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_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*)

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

(AbjadObject).**__format__**(*format_specification*='')

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

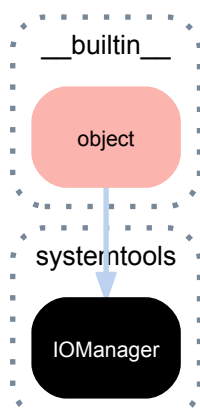
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

39.2.3 systemtools.IOManager



class `systemtools.IOManager`
 Manages Abjad IO.

Bases

- `__builtin__.object`

Static methods

`IOManager.clear_terminal()`
 Runs `clear` if OS is POSIX-compliant (UNIX / Linux / MacOS).
 Runs `cls` if OS is not POSIX-compliant (Windows).

```
>>> IOManager.clear_terminal()
```

Returns `none`.

`IOManager.count_function_calls(expr, global_context=None, local_context=None, fixed_point=True)`
 Counts function calls returned by `IOManager.profile_expr(expr)`.

Example 1. Function calls required to initialize note from string:

```
>>> systemtools.IOManager.count_function_calls(
...     "Note('c4')",
...     globals(),
...     )
9746
```

Example 2. Function calls required to initialize note from integers:

```
>>> systemtools.IOManager.count_function_calls(
...     "Note(-12, (1, 4))",
...     globals(),
...     )
170
```

Returns nonnegative integer.

`IOManager.ensure_directory_existence(directory)`
 Ensures existence of *directory*.

Returns `none`.

`IOManager.find_executable(name, flags=1)`
 Finds executable *name*.

Similar to Unix `which` command.

```
>>> IOManager.find_executable('python2.7')
['/usr/bin/python2.7']
```

Returns list of zero or more full paths to *name*.

`IOManager.get_last_output_file_name(output_directory_path=None)`
 Gets last output file name in output directory.

```
>>> systemtools.IOManager.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`.

`IOManager.get_next_output_file_name` (*file_extension='ly', output_directory_path=None*)
Gets next output file name in output directory.

```
>>> systemtools.IOManager.get_next_output_file_name()
'6223.ly'
```

Gets next output file name in Abjad output directory when *output_directory_path* is none.

Returns string.

`IOManager.insert_expr_into_lilypond_file` (*expr, tagline=False*)
Inserts *expr* into LilyPond file.

Returns LilyPond file.

`IOManager.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.

`IOManager.profile_expr` (*expr, sort_by='cumulative', line_count=12, strip_dirs=True, print_callers=False, print callees=False, global_context=None, local_context=None, print_to_terminal=True*)
Profiles *expr*.

```
>>> expr = 'Staff(scoretools.make_repeated_notes(8))'
>>> IOManager.profile_expr(expr)
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 'cumulative', '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.

`IOManager.run_lilypond` (*lilypond_file_name, lilypond_path=None*)
Runs LilyPond.

Returns none.

`IOManager.save_last_ly_as` (*file_path*)
Saves last LilyPond file created in Abjad as *file_path*.

```
>>> file_path = '/project/output/example-1.ly'
>>> IOManager.save_last_ly_as(file_path)
```

Returns none.

`IOManager.save_last_pdf_as(file_path)`
Saves last PDF created in Abjad as *file_path*.

```
>>> file_path = '/project/output/example-1.pdf'
>>> IOManager.save_last_pdf_as(file_path)
```

Returns none.

`IOManager.spawn_subprocess(command)`
Spawns subprocess and runs *command*.

Redirects stderr to stdout.

```
>>> command = 'echo "hellow world"'
>>> IOManager.spawn_subprocess(command)
hellow world
```

The function is basically a reimplementation of the deprecated `os.system()` using Python's `subprocess` module.

Returns integer result code.

`IOManager.view_last_log()`
Opens the LilyPond log file in operating system-specific text editor.

```
>>> systemtools.IOManager.view_last_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.

`IOManager.view_last_ly(target=-1)`
Opens the last LilyPond output file in text editor.

Example 1. Open the last LilyPond output file:

```
>>> systemtools.IOManager.view_last_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:

```
>>> systemtools.IOManager.view_last_ly(-2)
```

Returns none.

`IOManager.view_last_pdf(target=-1)`

Opens the last PDF generated by Abjad with `systemtools.pdf()`.

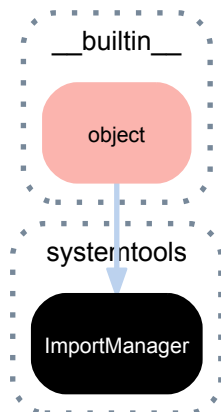
Opens the next-to-last PDF generated by Abjad with `systemtools.pdf(-2)`.

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.

39.2.4 systemtools.ImportManager



class `systemtools.ImportManager`
Imports structured packages.

Bases

- `__builtin__.object`

Static methods

`ImportManager.import_public_names_from_filesystem_path_into_namespace` (*path*,
namespace,
delete_systemtools=True,
***kwargs*)

Inspects the top level of *path*.

Finds `.py` modules in *path* and imports public functions from `.py` modules into *namespace*.

Finds packages in *path* and imports package names into *namespace*.

Does not import package content into *namespace*.

Does not inspect lower levels of *path*.

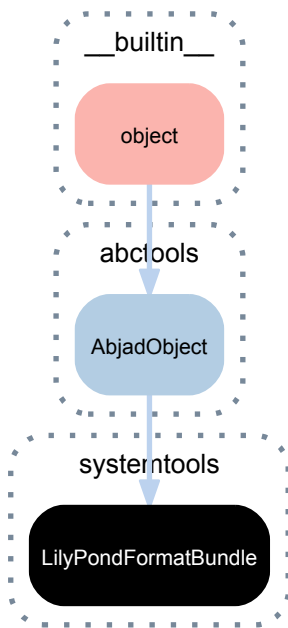
`ImportManager.import_structured_package` (*path*, *namespace*, *delete_systemtools=True*,
***kwargs*)

Imports 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.

39.2.5 systemtools.LilyPondFormatBundle



class `systemtools.LilyPondFormatBundle`
 LilyPond format bundle.

Transient class created to hold the collection of all format contributions generated on behalf of a single component.

Bases

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

Read-only properties

`LilyPondFormatBundle.after`
 After slot contributions.

Returns slot contributions object.

`LilyPondFormatBundle.before`
 Before slot contributions.

Returns slot contributions object.

`LilyPondFormatBundle.closing`
 Closing slot contributions.

Returns slot contributions object.

`LilyPondFormatBundle.context_settings`
 Context setting format contributions.

Returns list.

`LilyPondFormatBundle.grob_overrides`
 Grob override format contributions.

Returns list.

`LilyPondFormatBundle.grob_reverts`

Grob revert format contributions.

Returns list.

`LilyPondFormatBundle.opening`

Opening slot contributions.

Returns slot contributions object.

`LilyPondFormatBundle.right`

Right slot contributions.

Returns slot contributions object.

Methods

`LilyPondFormatBundle.alphabetize()`

Alphabetize format contributions in each slot.

Returns none.

`LilyPondFormatBundle.get(identifier)`

Get *identifier*.

Returns format contributions object or list.

`LilyPondFormatBundle.make_immutable()`

Make each slot immutable.

Returns none.

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

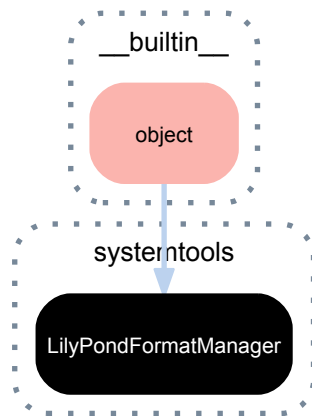
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

39.2.6 systemtools.LilyPondFormatManager



class `systemtools.LilyPondFormatManager`
 Manages LilyPond formatting logic.

Bases

- `__builtin__.object`

Static methods

`LilyPondFormatManager.bundle_format_contributions` (*component*)
 Gets all format contributions for *component*.

Returns LilyPond format bundle.

`LilyPondFormatManager.format_lilypond_attribute` (*attribute*)
 Formats LilyPond attribute according to Scheme formatting conventions.

Returns string.

`LilyPondFormatManager.format_lilypond_context_setting_in_with_block` (*name*,
value)

Formats LilyPond context setting *name* with *value* in LilyPond with-block.

Returns string.

`LilyPondFormatManager.format_lilypond_context_setting_inline` (*name*,
value, *context*=None)

Formats LilyPond context setting *name* with *value* in *context*.

Returns string.

`LilyPondFormatManager.format_lilypond_value` (*expr*)
 Formats LilyPond *expr* according to Scheme formatting conventions.

Returns string.

`LilyPondFormatManager.make_lilypond_override_string` (*grob_name*, *grob_attribute*,
grob_value, *context*
text_name=None,
is_once=False)

Makes Lilypond override string.

Does not include 'once'.

Returns string.

`LilyPondFormatManager.make_lilypond_revert_string` (*grob_name*, *grob_attribute*,
context_name=None)

Makes LilyPond revert string.

Returns string.

`LilyPondFormatManager.report_component_format_contributions` (*component*, *verbose=False*)

Reports *component* format contributions.

```
>>> staff = Staff("c'4 [ ( d'4 e'4 f'4 ] )")
>>> override(staff[0]).note_head.color = 'red'
```

```
>>> manager = systemtools.LilyPondFormatManager
>>> print manager.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.

`LilyPondFormatManager.report_spanner_format_contributions` (*spanner*)

Reports spanner format contributions for every leaf to which spanner attaches.

```
>>> staff = Staff("c8 d e f")
>>> spanner = spannertools.Beam()
>>> attach(spanner, staff[:])
```

```
>>> manager = systemtools.LilyPondFormatManager
>>> print manager.report_spanner_format_contributions(spanner)
c8 before: []
   after: []
   right: ['[']

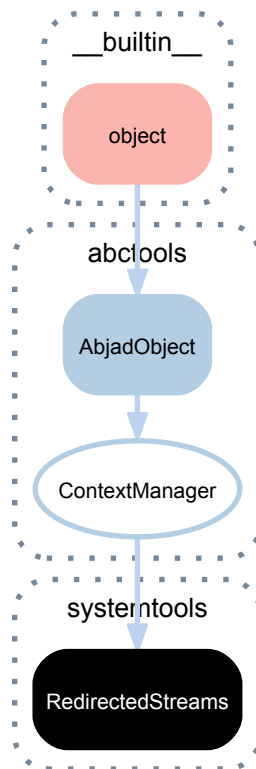
d8 before: []
   after: []
   right: []

e8 before: []
   after: []
   right: []

f8 before: []
   after: []
   right: [']']
```

Returns none or return string.

39.2.7 systemtools.RedirectedStreams



class systemtools.**RedirectedStreams** (*stdout=None, stderr=None*)
 A context manager for capturing stdout and stderr output.

```

>>> import StringIO
>>> string_io = StringIO.StringIO()
>>> with systemtools.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__()
 Enters redirected streams context manager.
 Returns none.

(AbjadObject).__eq__(expr)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

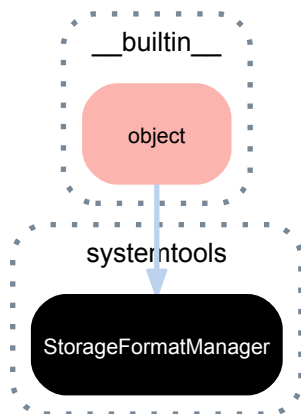
`RedirectedStreams.__exit__ (exc_type, exc_value, traceback)`
 Exits redirected streams context manager.
 Returns none.

`(AbjadObject).__format__ (format_specification='')`
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

`(AbjadObject).__ne__ (expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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

39.2.8 systemtools.StorageFormatManager



class `systemtools.StorageFormatManager`
 Manages Abjad object storage formats.

Bases

- `__builtin__.object`

Static methods

`StorageFormatManager.compare (object_one, object_two)`
 Compares *object_one* to *object_two*.
 Returns boolean.

`StorageFormatManager.format_one_value (value, is_indented=True, as_storage_format=True)`
 Formats one value.
 Returns list.

`StorageFormatManager.get_format_pieces (specification, as_storage_format=True)`
 Gets format pieces.

`StorageFormatManager.get_indentation_strings (is_indented)`
 Gets indentation strings.

`StorageFormatManager.get_input_argument_values (object_)`

Gets input argument values.

`StorageFormatManager.get_keyword_argument_dictionary (object_)`

Gets keyword argument dictionary.

`StorageFormatManager.get_keyword_argument_names (object_)`

Gets keyword argument names.

`StorageFormatManager.get_keyword_argument_values (object_)`

Gets keyword argument values.

`StorageFormatManager.get_positional_argument_dictionary (object_)`

Gets positional argument dictionary.

`StorageFormatManager.get_positional_argument_names (object_)`

Gets positional argument names.

`StorageFormatManager.get_positional_argument_values (object_)`

Gets positional argument values.

`StorageFormatManager.get_repr_format (object_)`

Gets interpreter representation format.

`StorageFormatManager.get_signature_keyword_argument_names (object_)`

Gets signature keyword argument names.

`StorageFormatManager.get_signature_positional_argument_names (object_)`

Gets signature positional argument names.

`StorageFormatManager.get_storage_format (object_)`

Gets storage format.

`StorageFormatManager.get_tools_package_name (object_)`

Gets tools-package name of *object_*.

```
>>> manager = systemtools.StorageFormatManager
>>> manager.get_tools_package_name(Note)
'scoretools'
```

`StorageFormatManager.get_tools_package_qualified_class_name (object_)`

Gets tools-package qualified class name of *object_*.

```
>>> manager = systemtools.StorageFormatManager
>>> manager.get_tools_package_qualified_class_name(Note)
'scoretools.Note'
```

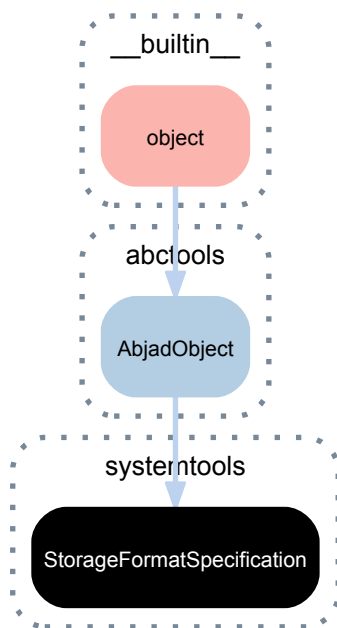
Returns string.

`StorageFormatManager.is_instance (object_)`

True when *object_* is instance. Otherwise false.

Returns boolean.

39.2.9 systemtools.StorageFormatSpecification



```
class systemtools.StorageFormatSpecification (instance=None,      body_text=None,
                                             is_bracketted=False,  is_indented=True,
                                             keyword_argument_names=None,  key-
                                             words_ignored_when_false=None,
                                             positional_argument_values=None,
                                             storage_format_pieces=None,
                                             tools_package_name=None)
```

Specifies the storage format of a given object.

Bases

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

Read-only properties

`StorageFormatSpecification.body_text`

Body text of storage specification.

Returns string.

`StorageFormatSpecification.instance`

Instance of storage specification.

Returns string.

`StorageFormatSpecification.is_bracketted`

True when storage specification is bracketted. Otherwise false.

Returns boolean.

`StorageFormatSpecification.is_indented`

True when storage format is indented. Otherwise false.

Returns boolean.

`StorageFormatSpecification.keyword_argument_names`

Keyword argument names of storage format.

Returns tuple.

`StorageFormatSpecification.keywords_ignored_when_false`
Keywords ignored when false.

Returns tuple.

`StorageFormatSpecification.positional_argument_values`
Positional argument values.

Returns tuple.

`StorageFormatSpecification.storage_format_pieces`
Storage format pieces.

Returns tuple.

`StorageFormatSpecification.tools_package_name`
Tools package name of storage format.

Returns string.

Special methods

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

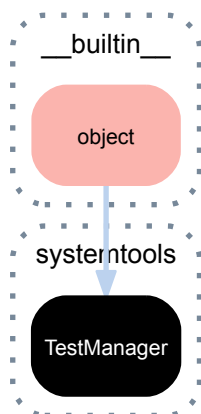
`(AbjadObject).__format__(format_specification='')`
Formats object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`StorageFormatSpecification.__makenew__(*args, **kwargs)`
Makes new storage format specification with optional *kwargs*.
Returns new storage format specification.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

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

39.2.10 systemtools.TestManager



class `systemtools.TestManager`
Manages test logic.

Bases

- `__builtin__.object`

Static methods

`TestManager.apply_additional_layout` (*lilypond_file*)
Configures multiple-voice rhythmic staves in *lilypond_file*.

Operates in place.

Returns none.

`TestManager.compare` (*string_1*, *string_2*)
Compares *string_1* to *string_2*.

Massage newlines.

Returns boolean.

`TestManager.get_current_function_name` ()
Gets current function name.

```
>>> def foo():
...     function_name = systemtools.TestManager.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.

`TestManager.read_test_output` (*full_file_name*, *current_function_name*)
Reads test output.

Returns list.

`TestManager.test_function_name_to_title_lines` (*test_function_name*)
Changes *test_function_name* to title lines.

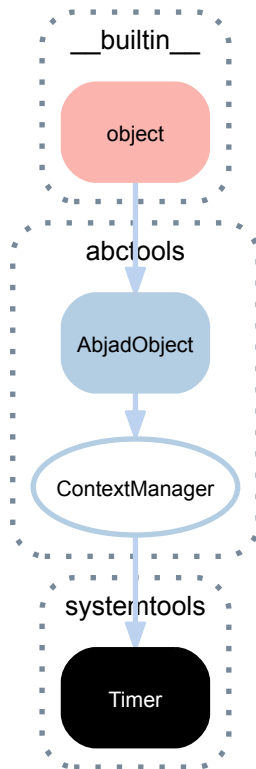
Returns list.

`TestManager.write_test_output` (*output, full_file_name, test_function_name, cache_ly=False, cache_pdf=False, go=False, render_pdf=False*)

Writes test output.

Returns none.

39.2.11 systemtools.Timer



class `systemtools.Timer`
A timing context manager:

```

>>> timer = systemtools.Timer()
>>> with timer:
...     for _ in xrange(1000000):
...         x = 1 + 1
...
>>> timer.elapsed_time
0.092742919921875
  
```

Bases

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

Read-only properties

`Timer.elapsed_time`

Elapsed time.

Return float or none.

`Timer.start_time`
Start time of timer.

Returns time.

`Timer.stop_time`
Stop time of timer.

Returns time.

Special methods

`Timer.__enter__()`
Enters context manager.

Returns none.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`Timer.__exit__(exc_type, exc_value, traceback)`
Exist context manager.

Returns none.

`(AbjadObject).__format__(format_specification='')`
Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

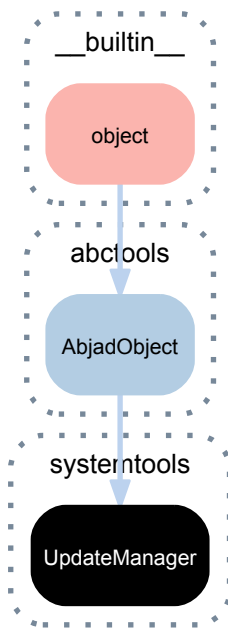
`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`
Gets interpreter representation of Abjad object.

Returns string.

39.2.12 systemtools.UpdateManager



class `systemtools.UpdateManager`

Update start offset, stop offsets and indicators everywhere in score.

Bases

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

Special methods

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

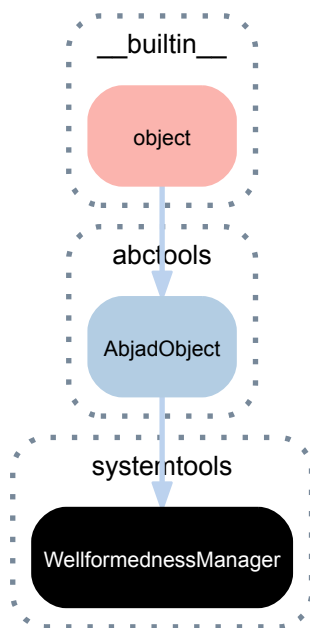
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

39.2.13 systemtools.WellformednessManager



class `systemtools.WellformednessManager` (*expr=None, allow_empty_containers=True*)
Wellformedness manager.

Bases

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

Methods

`WellformednessManager.check_beamed_quarter_notes()`

Checks to make sure there are no beamed quarter notes.

Returns violators and total.

`WellformednessManager.check_discontiguous_spanners()`

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*.

Returns violators and total.

`WellformednessManager.check_duplicate_ids()`

Checks to make sure there are no components with duplicated IDs.

Returns violators and total.

`WellformednessManager.check_empty_containers()`

Checks to make sure there are no empty containers in score.

Returns violators and total.

`WellformednessManager.check_intermarked_hairpins()`

Checks to make sure there are no hairpins in score with intervening dynamic marks.

Returns violators and total.

`WellformednessManager.check_misdurated_measures()`
 Checks to make sure there are no misdurated measures in score.
 Returns violators and total.

`WellformednessManager.check_misfilled_measures()`
 Checks that time signature duration equals measure contents duration for every measure.
 Returns violators and total.

`WellformednessManager.check_mispitched_ties()`
 Checks for mispitched notes. Does not check tied rests or skips.
 Returns violators and total.

`WellformednessManager.check_misrepresented_flags()`
 Checks to make sure there are no misrepresented flags in score.
 Returns violators and total.

`WellformednessManager.check_missing_parents()`
 Checks to make sure there are no components in score with missing parent.
 Returns violators and total.

`WellformednessManager.check_nested_measures()`
 Checks to make sure there are no nested measures in score.
 Returns violators and total.

`WellformednessManager.check_overlapping_beams()`
 Checks to make sure there are no overlapping beams in score.
 Returns violators and total.

`WellformednessManager.check_overlapping_glissandi()`
 Checks to make sure there are no overlapping glissandi in score.
 Returns violators and total.

`WellformednessManager.check_overlapping_octavation_spanners()`
 Checks to make sure there are no overlapping octavation spanners in score.
 Returns violators and total.

`WellformednessManager.check_short_hairpins()`
 Checks to make sure that hairpins span at least two leaves.
 Returns violators and total.

Special methods

`WellformednessManager.__call__()`
 Calls all wellformedness checks on *expr*.
 Returns something.

`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

39.3 Functions

39.3.1 systemtools.requires

`systemtools.requires(*tests)`

Function decorator to require input parameter *tests*.

```
>>> @systemtools.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.

39.3.2 systemtools.run_abjad

`systemtools.run_abjad()`

Runs Abjad.

Returns none.

A

- AbjadAPIGenerator (class in abjad.tools.documentationtools.AbjadAPIGenerator), 1384
- AbjadBookProcessor (class in abjad.tools.abjadbooktools.AbjadBookProcessor), 1323
- AbjadBookScript (class in abjad.tools.abjadbooktools.AbjadBookScript), 1325
- AbjadConfiguration (class in abjad.tools.systemtools.AbjadConfiguration), 1593
- AbjadObject (class in abjad.tools.abctools.AbjadObject), 1320
- AbjDevScript (class in abjad.tools.developerscripttools.AbjDevScript), 1340
- AbjGrepScript (class in abjad.tools.developerscripttools.AbjGrepScript), 1343
- Accidental (class in abjad.tools.pitchtools.Accidental), 512
- Accordion (class in abjad.tools.instrumenttools.Accordion), 187
- add_bell_music_to_score() (in module abjad.demos.part.add_bell_music_to_score), 1312
- add_string_music_to_score() (in module abjad.demos.part.add_string_music_to_score), 1312
- add_terminal_newlines() (in module abjad.tools.stringtools.add_terminal_newlines), 1167
- all_are_assignable_integers() (in module abjad.tools.sequencetools.all_are_assignable_integers), 1037
- all_are_equal() (in module abjad.tools.sequencetools.all_are_equal), 1037
- all_are_integer_equivalent_exprs() (in module abjad.tools.sequencetools.all_are_integer_equivalent_exprs), 1037
- all_are_integer_equivalent_numbers() (in module abjad.tools.sequencetools.all_are_integer_equivalent_numbers), 1038
- all_are_nonnegative_integer_equivalent_numbers() (in module abjad.tools.sequencetools.all_are_nonnegative_integer_equivalent_numbers), 1038
- all_are_nonnegative_integer_powers_of_two() (in module abjad.tools.sequencetools.all_are_nonnegative_integer_powers_of_two), 1038
- all_are_nonnegative_integers() (in module abjad.tools.sequencetools.all_are_nonnegative_integers), 1038
- all_are_numbers() (in module abjad.tools.sequencetools.all_are_numbers), 1039
- all_are_pairs() (in module abjad.tools.sequencetools.all_are_pairs), 1039
- all_are_pairs_of_types() (in module abjad.tools.sequencetools.all_are_pairs_of_types), 1039
- all_are_positive_integer_equivalent_numbers() (in module abjad.tools.sequencetools.all_are_positive_integer_equivalent_numbers), 1040
- all_are_positive_integers() (in module abjad.tools.sequencetools.all_are_positive_integers), 1040
- all_are_unequal() (in module abjad.tools.sequencetools.all_are_unequal), 1040
- AltoFlute (class in abjad.tools.instrumenttools.AltoFlute), 191
- AltoSaxophone (class in abjad.tools.instrumenttools.AltoSaxophone), 194
- AltoTrombone (class in abjad.tools.instrumenttools.AltoTrombone), 197
- AltoVoice (class in abjad.tools.instrumenttools.AltoVoice), 200
- Annotation (class in abjad.tools.indicatortools.Annotation), 147
- append_spacer_skip_to_underfull_measure() (in module abjad.tools.scoretools.append_spacer_skip_to_underfull_measure), 989
- append_spacer_skips_to_underfull_measures_in_expr()

(in module abjad.tools.lilypondfiletools.AttributedBlock),
 jad.tools.scoretools.append_spacer_skips_to_underfull_measures_in_expr(),
 989

B

apply_accidental_to_named_pitch() (in module abjad.tools.pitchtools.apply_accidental_to_named_pitch),
 633

apply_bowing_marks() (in module abjad.demos.part.apply_bowing_marks),
 1312

apply_dynamics() (in module abjad.demos.part.apply_dynamics), 1312

apply_expressive_marks() (in module abjad.demos.part.apply_expressive_marks),
 1312

apply_final_bar_lines() (in module abjad.demos.part.apply_final_bar_lines),
 1312

apply_full_measure_tuplets_to_contents_of_measures_in_expr() (in module abjad.tools.scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr),
 990

apply_page_breaks() (in module abjad.demos.part.apply_page_breaks), 1312

apply_rehearsal_marks() (in module abjad.demos.part.apply_rehearsal_marks),
 1313

are_relatively_prime() (in module abjad.tools.mathtools.are_relatively_prime),
 442

arg_to_bidirectional_direction_string() (in module abjad.tools.stringtools.arg_to_bidirectional_direction_string),
 1167

arg_to_bidirectional_lilypond_symbol() (in module abjad.tools.stringtools.arg_to_bidirectional_lilypond_symbol),
 1167

arg_to_tridirectional_direction_string() (in module abjad.tools.stringtools.arg_to_tridirectional_direction_string),
 1168

arg_to_tridirectional_lilypond_symbol() (in module abjad.tools.stringtools.arg_to_tridirectional_lilypond_symbol),
 1168

arg_to_tridirectional_ordinal_constant() (in module abjad.tools.stringtools.arg_to_tridirectional_ordinal_constant),
 1169

arithmetic_mean() (in module abjad.tools.mathtools.arithmetic_mean),
 442

Articulation (class in abjad.tools.indicatortools.Aarticulation), 149

AssignabilityError (class in abjad.tools.exceptiontools.AssignabilityError),
 1519

attach() (in module abjad.tools.topleveltools.attach),
 1297

AttackPointOptimizer (class in abjad.tools.quantizationtools.AttackPointOptimizer),
 647

AttributedBlock (class in abjad.tools.lilypondfiletools.AttributedBlock),
 373

BaritoneSaxophone (class in abjad.tools.instrumenttools.BaritoneSaxophone),
 203

BaritoneVoice (class in abjad.tools.instrumenttools.BaritoneVoice),
 206

BarLine (class in abjad.tools.indicatortools.BarLine),
 151

BaseResidueClass (class in abjad.tools.sievetools.BaseResidueClass),
 1077

BassClarinet (class in abjad.tools.instrumenttools.BassClarinet),
 209

BassFlute (class in abjad.tools.instrumenttools.BassFlute), 212

Bassoon (class in abjad.tools.instrumenttools.Bassoon),
 224

BassSaxophone (class in abjad.tools.instrumenttools.BassSaxophone),
 215

BassTrombone (class in abjad.tools.instrumenttools.BassTrombone),
 218

BassVoice (class in abjad.tools.instrumenttools.BassVoice), 221

Beam (class in abjad.tools.spannertools.Beam), 1085

BeatwiseQSchema (class in abjad.tools.quantizationtools.BeatwiseQSchema),
 650

BeatwiseQSchemaItem (class in abjad.tools.quantizationtools.BeatwiseQSchemaItem),
 655

BeatwiseQTarget (class in abjad.tools.quantizationtools.BeatwiseQTarget),
 657

BenchmarkScoreMaker (class in abjad.tools.systemtools.BenchmarkScoreMaker),
 1597

BendAfter (class in abjad.tools.indicatortools.BendAfter), 152

binomial_coefficient() (in module abjad.tools.mathtools.binomial_coefficient),
 442

BookBlock (class in abjad.tools.lilypondfiletools.BookBlock),
 382

BookpartBlock (class in abjad.tools.lilypondfiletools.BookpartBlock),
 385

BoundedObject (class in abjad.tools.mathtools.BoundedObject), 425

BuildApiScript (class in abjad.tools.developerscripttools.BuildApiScript),

1623

jad.tools.pitchtools.contains_subsegment), 634	jad.tools.datastructuretools.CyclicList), 37
Context (class in abjad.tools.scoretools.Context), 863	CyclicMatrix (class in ab- jad.tools.datastructuretools.CyclicMatrix), 40
ContextBlock (class in ab- jad.tools.lilypondfiletools.ContextBlock), 387	CyclicPayloadTree (class in ab- jad.tools.datastructuretools.CyclicPayloadTree), 43
ContextManager (class in ab- jad.tools.abctools.ContextManager), 1317	CyclicTuple (class in ab- jad.tools.datastructuretools.CyclicTuple), 57
ContextSpeccedMusic (class in ab- jad.tools.lilypondparsertools.ContextSpeccedMusic), 1540	D
contextualize() (in module ab- jad.tools.topleveltools.contextualize), 1297	DateTimeToken (class in ab- jad.tools.lilypondfiletools.DateTimeToken), 390
ContiguousSelection (class in ab- jad.tools.selectiontools.ContiguousSelection), 1007	Decrescendo (class in ab- jad.tools.spannertools.Decrescendo), 1098
Contrabass (class in ab- jad.tools.instrumenttools.Contrabass), 239	Descendants (class in ab- jad.tools.selectiontools.Descendants), 1010
ContrabassClarinet (class in ab- jad.tools.instrumenttools.ContrabassClarinet), 242	detach() (in module abjad.tools.topleveltools.detach), 1297
ContrabassFlute (class in ab- jad.tools.instrumenttools.ContrabassFlute), 245	DeveloperScript (class in ab- jad.tools.developerscripttools.DeveloperScript), 1335
Contrabassoon (class in ab- jad.tools.instrumenttools.Contrabassoon), 251	difference_series() (in module ab- jad.tools.mathtools.difference_series), 444
ContrabassSaxophone (class in ab- jad.tools.instrumenttools.ContrabassSaxophone), 248	DirectoryScript (class in ab- jad.tools.developerscripttools.DirectoryScript), 1338
count_length_two_runs_in_sequence() (in module ab- jad.tools.sequencetools.count_length_two_runs_in_sequence), 1040	DiscardingGraceHandler (class in ab- jad.tools.quantizationtools.DiscardingGraceHandler), 671
CountLinewidthsScript (class in ab- jad.tools.developerscripttools.CountLinewidthsScript), 1350	DistanceHeuristic (class in ab- jad.tools.quantizationtools.DistanceHeuristic), 672
CountToolsScript (class in ab- jad.tools.developerscripttools.CountToolsScript), 1353	divide_number_by_ratio() (in module ab- jad.tools.mathtools.divide_number_by_ratio), 444
create_pitch_contour_reservoir() (in module ab- jad.demos.part.create_pitch_contour_reservoir), 1313	divide_sequence_elements_by_greatest_common_divisor() (in module ab- jad.tools.sequencetools.divide_sequence_elements_by_greatest), 1041
Crescendo (class in ab- jad.tools.spannertools.Crescendo), 1093	DivisionBurnishedTaleaRhythmMaker (class in ab- jad.tools.rhythmmakertools.DivisionBurnishedTaleaRhythmMa 751
cumulative_products() (in module ab- jad.tools.mathtools.cumulative_products), 443	DivisionIncisedNoteRhythmMaker (class in ab- jad.tools.rhythmmakertools.DivisionIncisedNoteRhythmMaker, 756
cumulative_signed_weights() (in module ab- jad.tools.mathtools.cumulative_signed_weights), 443	DivisionIncisedRestRhythmMaker (class in ab- jad.tools.rhythmmakertools.DivisionIncisedRestRhythmMaker), 761
cumulative_sums() (in module ab- jad.tools.mathtools.cumulative_sums), 443	DivisionIncisedRhythmMaker (class in ab- jad.tools.rhythmmakertools.DivisionIncisedRhythmMaker), 742
cumulative_sums_pairwise() (in module ab- jad.tools.mathtools.cumulative_sums_pairwise), 443	divisors() (in module abjad.tools.mathtools.divisors), 444
CyclicList (class in ab-	

Documenter (class in abjad.tools.documentationtools.Documenter), 1390	fill_measures_in_expr_with_repeated_notes() (in module abjad.tools.scoretools.fill_measures_in_expr_with_repeated_notes), 991
durate_pitch_contour_reservoir() (in module abjad.demos.part.durate_pitch_contour_reservoir), 1313	fill_measures_in_expr_with_time_signature_denominator_notes() (in module abjad.tools.scoretools.fill_measures_in_expr_with_time_signature_denominator_notes), 991
DuratedComplexBeam (class in abjad.tools.spannertools.DuratedComplexBeam), 1103	FixedDurationContainer (class in abjad.tools.scoretools.FixedDurationContainer), 870
Duration (class in abjad.tools.durationtools.Duration), 111	FixedDurationTuplet (class in abjad.tools.scoretools.FixedDurationTuplet), 876
Dynamic (class in abjad.tools.indicatortools.Dynamic), 161	flatten_sequence() (in module abjad.tools.sequencetools.flatten_sequence), 1041
DynamicTextSpanner (class in abjad.tools.spannertools.DynamicTextSpanner), 1108	flatten_sequence_at_indices() (in module abjad.tools.sequencetools.flatten_sequence_at_indices), 1041
E	Flute (class in abjad.tools.instrumenttools.Flute), 257
edit_bass_voice() (in module abjad.demos.part.edit_bass_voice), 1313	format_input_lines_as_doc_string() (in module abjad.tools.stringtools.format_input_lines_as_doc_string), 1169
edit_cello_voice() (in module abjad.demos.part.edit_cello_voice), 1313	format_input_lines_as_regression_test() (in module abjad.tools.stringtools.format_input_lines_as_regression_test), 1170
edit_first_violin_voice() (in module abjad.demos.part.edit_first_violin_voice), 1313	fraction_to_proper_fraction() (in module abjad.tools.mathtools.fraction_to_proper_fraction), 445
edit_second_violin_voice() (in module abjad.demos.part.edit_second_violin_voice), 1313	FrenchHorn (class in abjad.tools.instrumenttools.FrenchHorn), 260
edit_viola_voice() (in module abjad.demos.part.edit_viola_voice), 1313	FunctionCrawler (class in abjad.tools.documentationtools.FunctionCrawler), 1392
EnglishHorn (class in abjad.tools.instrumenttools.EnglishHorn), 254	FunctionDocumenter (class in abjad.tools.documentationtools.FunctionDocumenter), 1393
EqualDivisionRhythmMaker (class in abjad.tools.rhythmmakertools.EqualDivisionRhythmMaker), 765	
EvenRunRhythmMaker (class in abjad.tools.rhythmmakertools.EvenRunRhythmMaker), 768	
extend_measures_in_expr_and_apply_full_measure_tuplets() (in module abjad.tools.scoretools.extend_measures_in_expr_and_apply_full_measure_tuplets), 990	G
ExtraSpannerError (class in abjad.tools.exceptiontools.ExtraSpannerError), 1520	get_developer_script_classes() (in module abjad.tools.developerscripttools.get_developer_script_classes), 1372
F	get_indices_of_sequence_elements_equal_to_true() (in module abjad.tools.sequencetools.get_indices_of_sequence_elements_equal_to_true), 1042
factors() (in module abjad.tools.mathtools.factors), 445	get_measure_that_starts_with_container() (in module abjad.tools.scoretools.get_measure_that_starts_with_container), 991
fill_measures_in_expr_with_full_measure_spacer_skips() (in module abjad.tools.scoretools.fill_measures_in_expr_with_full_measure_spacer_skips), 990	get_measure_that_stops_with_container() (in module abjad.tools.scoretools.get_measure_that_stops_with_container), 991
fill_measures_in_expr_with_minimal_number_of_notes() (in module abjad.tools.scoretools.fill_measures_in_expr_with_minimal_number_of_notes), 991	get_named_pitch_from_pitch_carrier() (in module abjad.tools.pitchtools.get_named_pitch_from_pitch_carrier), 991

634		jad.tools.documentationtools.GraphvizNode),
get_next_measure_from_component()	(in module ab-	1409
jad.tools.scoretools.get_next_measure_from_component()	jad.tools.scoretools.get_next_measure_from_component()	jad.tools.documentationtools.GraphvizObject (class in ab-
992		jad.tools.documentationtools.GraphvizObject),
get_numbered_pitch_class_from_pitch_carrier()		1373
(in module ab-	GraphvizSubgraph (class in ab-	
jad.tools.pitchtools.get_numbered_pitch_class_from_pitch_carrier())	jad.tools.pitchtools.get_numbered_pitch_class_from_pitch_carrier()	jad.tools.documentationtools.GraphvizSubgraph),
635		1413
get_one_indexed_measure_number_in_expr()	greatest_common_divisor() (in module ab-	
(in module ab-	jad.tools.mathtools.greatest_common_divisor),	
jad.tools.scoretools.get_one_indexed_measure_number_in_expr(),		
992		greatest_multiple_less_equal() (in module ab-
get_previous_measure_from_component()	jad.tools.mathtools.greatest_multiple_less_equal),	
(in module ab-		446
jad.tools.scoretools.get_previous_measure_from_component(),	greatest_power_of_two_less_equal() (in module ab-	
992		jad.tools.mathtools.greatest_power_of_two_less_equal),
get_sequence_degree_of_rotational_symmetry()		447
(in module ab-	GroupedRhythmicStavesScoreTemplate (class in ab-	
jad.tools.sequencetools.get_sequence_degree_of_rotational_symmetry())	jad.tools.sequencetools.get_sequence_degree_of_rotational_symmetry()	jad.tools.sequencetools.GroupedRhythmicStavesScoreTemplate
1042		1177
get_sequence_element_at_cyclic_index()	GroupedStavesScoreTemplate (class in ab-	
(in module ab-	jad.tools.templatetools.GroupedStavesScoreTemplate),	
jad.tools.sequencetools.get_sequence_element_at_cyclic_index(),		
1042		GuileProxy (class in ab-
get_sequence_elements_at_indices()	(in module ab-	jad.tools.lilypondparsertools.GuileProxy),
jad.tools.sequencetools.get_sequence_elements_at_indices(),		541
1043		Guitar (class in abjad.tools.instrumenttools.Guitar), 266
get_sequence_elements_frequency_distribution()		
(in module ab-	H	
jad.tools.sequencetools.get_sequence_elements_frequency_distribution(),	Harp (class in abjad.tools.instrumenttools.Harp), 269	
1043		1114
get_sequence_period_of_rotation()	(in module ab-	Harp (class in abjad.tools.instrumenttools.Harp), 269
jad.tools.sequencetools.get_sequence_period_of_rotation(),		
1043		Harpsichord (class in ab-
get_shared_numeric_sign()	(in module ab-	jad.tools.instrumenttools.Harpsichord),
jad.tools.mathtools.get_shared_numeric_sign(),		272
445		HeaderBlock (class in ab-
Glissando (class in abjad.tools.spannertools.Glissando),		jad.tools.lilypondfiletools.HeaderBlock),
1111		391
Glockenspiel (class in ab-	Heuristic (class in ab-	
jad.tools.instrumenttools.Glockenspiel),		jad.tools.quantizationtools.Heuristic), 650
263		HiddenStaffSpanner (class in ab-
GraceContainer (class in ab-		jad.tools.spannertools.HiddenStaffSpanner),
jad.tools.scoretools.GraceContainer), 893		1119
GraceHandler (class in ab-	HorizontalBracketSpanner (class in ab-	
jad.tools.quantizationtools.GraceHandler),		jad.tools.spannertools.HorizontalBracketSpanner),
649		1122
GrandStaff (class in abjad.tools.scoretools.GrandStaff),	HTMLOutputFormat (class in ab-	
899		jad.tools.abjadbooktools.HTMLOutputFormat),
graph() (in module abjad.tools.topleveltools.graph),		1329
1297		I
GraphvizEdge (class in ab-	ImportManager (class in ab-	
jad.tools.documentationtools.GraphvizEdge),		jad.tools.systemtools.ImportManager),
1395		1604
GraphvizGraph (class in ab-	ImpreciseTempoError (class in ab-	
jad.tools.documentationtools.GraphvizGraph),		jad.tools.exceptiontools.ImpreciseTempoError),
1397		1521
GraphvizNode (class in ab-		

IncisedRhythmMaker	(class in abjad.tools.rhythmmakertools.IncisedRhythmMaker), 745	IntervalClassVector), 523
IntervalSegment	(class in abjad.tools.pitchtools.IntervalSegment), 527	
increase_sequence_elements_at_indices_by_addenda()	(in module abjad.tools.sequencetools.increase_sequence_elements_at_indices_by_addenda), 1043	IntervalSet (class in abjad.tools.pitchtools.IntervalSet), 527
increase_sequence_elements_cyclically_by_addenda()	(in module abjad.tools.sequencetools.increase_sequence_elements_cyclically_by_addenda), 1044	IntervalVector (class in abjad.tools.pitchtools.IntervalVector), 535
IndicatorExpression	(class in abjad.tools.indicatorertools.IndicatorExpression), 163	inventory_aggregate_subsets() (in module abjad.tools.pitchtools.inventory_aggregate_subsets), 636
Infinity	(class in abjad.tools.mathtools.Infinity), 427	IOManager (class in abjad.tools.systemtools.IOManager), 1600
InheritanceGraph	(class in abjad.tools.documentationtools.InheritanceGraph), 1422	is_assignable_integer() (in module abjad.tools.mathtools.is_assignable_integer), 448
insert_and_transpose_nested_subruns_in_pitch_class_number_list()	(in module abjad.tools.pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list), 635	is_dash_case_file_name() (in module abjad.tools.stringtools.is_dash_case_file_name), 1171
InspectionAgent	(class in abjad.tools.agenttools.InspectionAgent), 3	is_dash_case_string() (in module abjad.tools.stringtools.is_dash_case_string), 1171
instantiate_pitch_and_interval_test_collection()	(in module abjad.tools.pitchtools.instantiate_pitch_and_interval_test_collection), 636	is_dotted_integer() (in module abjad.tools.mathtools.is_dotted_integer), 448
Instrument	(class in abjad.tools.instrumenttools.Instrument), 275	is_fraction_equivalent_pair() (in module abjad.tools.sequencetools.is_fraction_equivalent_pair), 1045
InstrumentationSpecifier	(class in abjad.tools.instrumenttools.InstrumentationSpecifier), 281	is_integer_equivalent_expr() (in module abjad.tools.mathtools.is_integer_equivalent_expr), 449
InstrumentInventory	(class in abjad.tools.instrumenttools.InstrumentInventory), 277	is_integer_equivalent_n_tuple() (in module abjad.tools.sequencetools.is_integer_equivalent_n_tuple), 1044
integer_equivalent_number_to_integer()	(in module abjad.tools.mathtools.integer_equivalent_number_to_integer), 447	is_integer_equivalent_number() (in module abjad.tools.mathtools.is_integer_equivalent_number), 449
integer_to_base_k_tuple()	(in module abjad.tools.mathtools.integer_to_base_k_tuple), 447	is_integer_equivalent_pair() (in module abjad.tools.sequencetools.is_integer_equivalent_pair), 1045
integer_to_binary_string()	(in module abjad.tools.mathtools.integer_to_binary_string), 448	is_integer_equivalent_singleton() (in module abjad.tools.sequencetools.is_integer_equivalent_singleton), 1045
interlace_sequences()	(in module abjad.tools.sequencetools.interlace_sequences), 1044	is_integer_n_tuple() (in module abjad.tools.sequencetools.is_integer_n_tuple), 1045
Interval	(class in abjad.tools.pitchtools.Interval), 491	is_integer_pair() (in module abjad.tools.sequencetools.is_integer_pair), 1045
IntervalClass	(class in abjad.tools.pitchtools.IntervalClass), 494	is_integer_singleton() (in module abjad.tools.sequencetools.is_integer_singleton), 1046
IntervalClassSegment	(class in abjad.tools.pitchtools.IntervalClassSegment), 515	is_lower_camel_case_string() (in module abjad.tools.stringtools.is_lower_camel_case_string), 1171
IntervalClassSet	(class in abjad.tools.pitchtools.IntervalClassSet), 519	is_monotonically_decreasing_sequence() (in module abjad.tools.sequencetools.is_monotonically_decreasing_sequence), 1046
IntervalClassVector	(class in abjad.tools.pitchtools.IntervalClassVector), 519	

1046	jad.tools.stringtools.is_snake_case_string(),
is_monotonically_increasing_sequence()	1172
(in module ab-	is_space_delimited_lowercase_string() (in module ab-
jad.tools.sequencetools.is_monotonically_increasing_sequence()),	jad.tools.stringtools.is_space_delimited_lowercase_string()),
1046	1172
is_n_tuple() (in module ab-	is_strictly_decreasing_sequence() (in module ab-
jad.tools.sequencetools.is_n_tuple), 1047	jad.tools.sequencetools.is_strictly_decreasing_sequence),
is_negative_integer() (in module ab-	1049
jad.tools.mathtools.is_negative_integer),	is_strictly_increasing_sequence() (in module ab-
449	jad.tools.sequencetools.is_strictly_increasing_sequence),
is_nonnegative_integer() (in module ab-	1050
jad.tools.mathtools.is_nonnegative_integer),	is_upper_camel_case_string() (in module ab-
450	jad.tools.stringtools.is_upper_camel_case_string),
is_nonnegative_integer_equivalent_number()	1172
(in module ab-	IsAtSoundingPitch (class in ab-
jad.tools.mathtools.is_nonnegative_integer_equivalent_number()),	jad.tools.indicator tools.IsAtSoundingPitch),
450	164
is_nonnegative_integer_power_of_two()	IsUnpitched (class in ab-
(in module ab-	jad.tools.indicator tools.IsUnpitched), 165
jad.tools.mathtools.is_nonnegative_integer_power_of_two()),	iterate_twice() (in module abjad.tools.topleveltools.iterate),
450	1298
is_null_tuple() (in module ab-	iterate_named_pitch_pairs_in_expr() (in module ab-
jad.tools.sequencetools.is_null_tuple),	jad.tools.pitchtools.iterate_named_pitch_pairs_in_expr),
1047	636
is_pair() (in module abjad.tools.sequencetools.is_pair),	iterate_out_of_range_notes_and_chords()
1047	(in module ab-
is_permutation() (in module ab-	jad.tools.instrumenttools.iterate_out_of_range_notes_and_chords),
jad.tools.sequencetools.is_permutation),	353
1048	iterate_sequence_cyclically() (in module ab-
is_positive_integer() (in module ab-	jad.tools.sequencetools.iterate_sequence_cyclically),
jad.tools.mathtools.is_positive_integer),	1050
451	iterate_sequence_cyclically_from_start_to_stop()
is_positive_integer_equivalent_number()	(in module ab-
(in module ab-	jad.tools.sequencetools.iterate_sequence_cyclically_from_start_to_stop),
jad.tools.mathtools.is_positive_integer_equivalent_number()),	451
451	iterate_sequence_forward_and_backward_nonoverlapping()
is_positive_integer_power_of_two() (in module ab-	(in module ab-
jad.tools.mathtools.is_positive_integer_power_of_two),	jad.tools.sequencetools.iterate_sequence_forward_and_backward_nonoverlapping),
451	1051
is_repetition_free_sequence() (in module ab-	iterate_sequence_forward_and_backward_overlapping()
jad.tools.sequencetools.is_repetition_free_sequence),	(in module ab-
1048	jad.tools.sequencetools.iterate_sequence_forward_and_backward_overlapping),
is_restricted_growth_function() (in module ab-	1051
jad.tools.sequencetools.is_restricted_growth_function()),	iterate_sequence_nwise_cyclic() (in module ab-
1048	jad.tools.sequencetools.iterate_sequence_nwise_cyclic),
is_singleton() (in module ab-	1051
jad.tools.sequencetools.is_singleton), 1049	iterate_sequence_nwise_strict() (in module ab-
is_snake_case_file_name() (in module ab-	jad.tools.sequencetools.iterate_sequence_nwise_strict),
jad.tools.stringtools.is_snake_case_file_name),	1052
1171	iterate_sequence_nwise_wrapped() (in module ab-
is_snake_case_file_name_with_extension()	jad.tools.sequencetools.iterate_sequence_nwise_wrapped),
(in module ab-	1052
jad.tools.stringtools.is_snake_case_file_name_with_extension()),	iterate_sequence_pairwise_cyclic() (in module ab-
1171	jad.tools.sequencetools.iterate_sequence_pairwise_cyclic),
is_snake_case_package_name() (in module ab-	1052
jad.tools.stringtools.is_snake_case_package_name()),	iterate_sequence_pairwise_strict() (in module ab-
1172	jad.tools.sequencetools.iterate_sequence_pairwise_strict),
is_snake_case_string() (in module ab-	1053

[iterate_sequence_pairwise_wrapped\(\)](#) (in module `abjad.tools.sequencetools`), 1053
[IterationAgent](#) (class in `abjad.tools.agenttools`), 9
J
[JobHandler](#) (class in `abjad.tools.quantizationtools`), 651
[join_subsequences\(\)](#) (in module `abjad.tools.sequencetools`), 1053
[join_subsequences_by_sign_of_subsequence_elements\(\)](#) (in module `abjad.tools.sequencetools`), 1053
K
[KeySignature](#) (class in `abjad.tools.indicator`), 166
L
[label_leaves_in_expr_with_leaf_depth\(\)](#) (in module `abjad.tools.labeltools`), 360
[label_leaves_in_expr_with_leaf_duration\(\)](#) (in module `abjad.tools.labeltools`), 361
[label_leaves_in_expr_with_leaf_durations\(\)](#) (in module `abjad.tools.labeltools`), 361
[label_leaves_in_expr_with_leaf_indices\(\)](#) (in module `abjad.tools.labeltools`), 362
[label_leaves_in_expr_with_leaf_numbers\(\)](#) (in module `abjad.tools.labeltools`), 362
[label_leaves_in_expr_with_named_interval_classes\(\)](#) (in module `abjad.tools.labeltools`), 363
[label_leaves_in_expr_with_named_intervals\(\)](#) (in module `abjad.tools.labeltools`), 363
[label_leaves_in_expr_with_numbered_interval_classes\(\)](#) (in module `abjad.tools.labeltools`), 363
[label_leaves_in_expr_with_numbered_intervals\(\)](#) (in module `abjad.tools.labeltools`), 370
[label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes\(\)](#) (in module `abjad.tools.labeltools`), 364
[label_leaves_in_expr_with_pitch_class_numbers\(\)](#) (in module `abjad.tools.labeltools`), 364
[label_leaves_in_expr_with_pitch_numbers\(\)](#) (in module `abjad.tools.labeltools`), 365
[label_leaves_in_expr_with_tuplet_depth\(\)](#) (in module `abjad.tools.labeltools`), 365
[label_leaves_in_expr_with_written_leaf_duration\(\)](#) (in module `abjad.tools.labeltools`), 366
[label_logical_ties_in_expr_with_logical_tie_duration\(\)](#) (in module `abjad.tools.labeltools`), 366
[label_logical_ties_in_expr_with_logical_tie_durations\(\)](#) (in module `abjad.tools.labeltools`), 367
[label_logical_ties_in_expr_with_written_logical_tie_duration\(\)](#) (in module `abjad.tools.labeltools`), 367
[label_notes_in_expr_with_note_indices\(\)](#) (in module `abjad.tools.labeltools`), 367
[label_vertical_moments_in_expr_with_interval_class_vectors\(\)](#) (in module `abjad.tools.labeltools`), 368
[label_vertical_moments_in_expr_with_named_intervals\(\)](#) (in module `abjad.tools.labeltools`), 369
[label_vertical_moments_in_expr_with_numbered_interval_classes\(\)](#) (in module `abjad.tools.labeltools`), 369
[label_vertical_moments_in_expr_with_numbered_intervals\(\)](#) (in module `abjad.tools.labeltools`), 369
[label_vertical_moments_in_expr_with_numbered_pitch_classes\(\)](#) (in module `abjad.tools.labeltools`), 370

label_vertical_moments_in_expr_with_pitch_numbers() (in module abjad.tools.labeltools.label_vertical_moments_in_expr_with_pitch_numbers), 370	1547 LilyPondNameManager (class in abjad.tools.lilypondnametools.LilyPondNameManager), 1534
LaTeXOutputFormat (class in abjad.tools.abjadbooktools.LaTeXOutputFormat), 1330	LilyPondParser (class in abjad.tools.lilypondparsertools.LilyPondParser), 1550
LayoutBlock (class in abjad.tools.lilypondfiletools.LayoutBlock), 394	LilyPondParserError (class in abjad.tools.exceptiontools.LilyPondParserError), 1522
Leaf (class in abjad.tools.scoretools.Leaf), 845	LilyPondSettingNameManager (class in abjad.tools.lilypondnametools.LilyPondSettingNameManager), 1535
least_common_multiple() (in module abjad.tools.mathtools.least_common_multiple), 451	LilyPondSyntacticalDefinition (class in abjad.tools.lilypondparsertools.LilyPondSyntacticalDefinition), 1560
least_multiple_greater_equal() (in module abjad.tools.mathtools.least_multiple_greater_equal), 452	LilyPondVersionToken (class in abjad.tools.lilypondfiletools.LilyPondVersionToken), 402
least_power_of_two_greater_equal() (in module abjad.tools.mathtools.least_power_of_two_greater_equal), 452	Lineage (class in abjad.tools.selectiontools.Lineage), 1012
LilyPondCommand (class in abjad.tools.indicatortools.LilyPondCommand), 168	list_all_abjad_classes() (in module abjad.tools.documentationtools.list_all_abjad_classes), 1516
LilyPondComment (class in abjad.tools.indicatortools.LilyPondComment), 169	list_all_abjad_functions() (in module abjad.tools.documentationtools.list_all_abjad_functions), 1516
LilyPondDimension (class in abjad.tools.lilypondfiletools.LilyPondDimension), 397	list_named_pitches_in_expr() (in module abjad.tools.pitchtools.list_named_pitches_in_expr), 637
LilyPondDuration (class in abjad.tools.lilypondparsertools.LilyPondDuration), 1543	list_numbered_interval_numbers_pairwise() (in module abjad.tools.pitchtools.list_numbered_interval_numbers_pairwise), 638
LilyPondEvent (class in abjad.tools.lilypondparsertools.LilyPondEvent), 1544	list_numbered_inversion_equivalent_interval_classes_pairwise() (in module abjad.tools.pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise), 638
LilyPondFile (class in abjad.tools.lilypondfiletools.LilyPondFile), 398	list_octave_transpositions_of_pitch_carrier_within_pitch_range() (in module abjad.tools.pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range), 639
LilyPondFormatBundle (class in abjad.tools.systemtools.LilyPondFormatBundle), 1605	list_ordered_named_pitch_pairs_from_expr_1_to_expr_2() (in module abjad.tools.pitchtools.list_ordered_named_pitch_pairs_from_expr_1_to_expr_2), 640
LilyPondFormatManager (class in abjad.tools.systemtools.LilyPondFormatManager), 1607	list_pitch_numbers_in_expr() (in module abjad.tools.pitchtools.list_pitch_numbers_in_expr), 640
LilyPondFraction (class in abjad.tools.lilypondparsertools.LilyPondFraction), 1545	list_unordered_named_pitch_pairs_in_expr() (in module abjad.tools.pitchtools.list_unordered_named_pitch_pairs_in_expr), 640
LilyPondGrammarGenerator (class in abjad.tools.lilypondparsertools.LilyPondGrammarGenerator), 1546	
LilyPondGrobNameManager (class in abjad.tools.lilypondnametools.LilyPondGrobNameManager), 1533	
LilyPondLanguageToken (class in abjad.tools.lilypondfiletools.LilyPondLanguageToken), 401	LogicalTie (class in abjad.tools.selectiontools.LogicalTie), 1014
LilyPondLexicalDefinition (class in abjad.tools.lilypondparsertools.LilyPondLexicalDefinition), 1546	
	make_basic_lilypond_file() (in module abjad.tools.lilypondfiletools.make_basic_lilypond_file), 1546

M

[jad.tools.lilypondfiletools.make_basic_lilypond_file\(\)](#), [1309](#)
[411](#) [make_mozart_measure\(\)](#) (in module [ab-](#)
[make_big_centered_page_number_markup\(\)](#) [jad.demos.mozart.make_mozart_measure\(\)](#),
(in module [ab-](#) [1309](#)
[jad.tools.markuptools.make_big_centered_page_number_markup\(\)](#) (in module [ab-](#)
[423](#) [jad.demos.mozart.make_mozart_measure_corpus\(\)](#) (in module [ab-](#)
[make_blank_line_markup\(\)](#) (in module [ab-](#) [1309](#)
[jad.tools.markuptools.make_blank_line_markup\(\)](#) [make_mozart_score\(\)](#) (in module [ab-](#)
[423](#) [jad.demos.mozart.make_mozart_score\(\)](#),
[make_centered_title_markup\(\)](#) (in module [ab-](#) [1309](#)
[jad.tools.markuptools.make_centered_title_markup\(\)](#) [make_multimeasure_rests\(\)](#) (in module [ab-](#)
[423](#) [jad.tools.scoretools.make_multimeasure_rests\(\)](#),
[make_colored_text_spanner_with_nibs\(\)](#) [996](#)
(in module [ab-](#) [make_multiplied_quarter_notes\(\)](#) (in module [ab-](#)
[jad.tools.spannertools.make_colored_text_spanner_with_nibs\(\)](#) [jad.tools.scoretools.make_multiplied_quarter_notes\(\)](#),
[1164](#) [997](#)
[make_desordre_cell\(\)](#) (in module [ab-](#) [make_n_middle_c_centered_pitches\(\)](#) (in module [ab-](#)
[jad.demos.desordre.make_desordre_cell\(\)](#), [jad.tools.pitchtools.make_n_middle_c_centered_pitches\(\)](#),
[1305](#) [640](#)
[make_desordre_lilypond_file\(\)](#) (in module [ab-](#) [make_nested_tuplet\(\)](#) (in module [ab-](#)
[jad.demos.desordre.make_desordre_lilypond_file\(\)](#), [jad.demos.ferneyhough.make_nested_tuplet\(\)](#),
[1305](#) [1307](#)
[make_desordre_measure\(\)](#) (in module [ab-](#) [make_notes\(\)](#) (in module [ab-](#)
[jad.demos.desordre.make_desordre_measure\(\)](#), [jad.tools.scoretools.make_notes\(\)](#), [997](#)
[1305](#) [make_notes_with_multiplied_durations\(\)](#)
[make_desordre_pitches\(\)](#) (in module [ab-](#) (in module [ab-](#)
[jad.demos.desordre.make_desordre_pitches\(\)](#), [jad.tools.scoretools.make_notes_with_multiplied_durations\(\)](#),
[1305](#) [998](#)
[make_desordre_score\(\)](#) (in module [ab-](#) [make_part_lilypond_file\(\)](#) (in module [ab-](#)
[jad.demos.desordre.make_desordre_score\(\)](#), [jad.demos.part.make_part_lilypond_file\(\)](#),
[1305](#) [1314](#)
[make_desordre_staff\(\)](#) (in module [ab-](#) [make_percussion_note\(\)](#) (in module [ab-](#)
[jad.demos.desordre.make_desordre_staff\(\)](#), [jad.tools.scoretools.make_percussion_note\(\)](#),
[1305](#) [998](#)
[make_dynamic_spanner_below_with_nib_at_right\(\)](#) [make_piano_score_from_leaves\(\)](#) (in module [ab-](#)
(in module [ab-](#) [jad.tools.scoretools.make_piano_score_from_leaves\(\)](#),
[jad.tools.spannertools.make_dynamic_spanner_below_with_nib_at_right\(\)](#), [998](#)
[1164](#) [make_piano_sketch_score_from_leaves\(\)](#)
[make_empty_piano_score\(\)](#) (in module [ab-](#) (in module [ab-](#)
[jad.tools.scoretools.make_empty_piano_score\(\)](#), [jad.tools.scoretools.make_piano_sketch_score_from_leaves\(\)](#),
[993](#) [999](#)
[make_floating_time_signature_lilypond_file\(\)](#) [make_reference_manual_graphviz_graph\(\)](#)
(in module [ab-](#) (in module [ab-](#)
[jad.tools.lilypondfiletools.make_floating_time_signature_lilypond_file\(\)](#), [jad.tools.documentatointools.make_reference_manual_graphviz](#)
[411](#) [1516](#)
[make_leaves\(\)](#) (in module [ab-](#) [make_reference_manual_lilypond_file\(\)](#) (in module [ab-](#)
[jad.tools.scoretools.make_leaves\(\)](#), [993](#) [jad.tools.documentatointools.make_reference_manual_lilypond](#)
[make_leaves_from_talea\(\)](#) (in module [ab-](#) [1516](#)
[jad.tools.scoretools.make_leaves_from_talea\(\)](#), [make_repeated_notes\(\)](#) (in module [ab-](#)
[996](#) [jad.tools.scoretools.make_repeated_notes\(\)](#),
[make_ligeti_example_lilypond_file\(\)](#) (in module [ab-](#) [999](#)
[jad.tools.documentatointools.make_ligeti_example_lilypond_file\(\)](#), [make_repeated_notes_from_time_signature\(\)](#)
[1516](#) (in module [ab-](#)
[make_lilypond_file\(\)](#) (in module [ab-](#) [jad.tools.scoretools.make_repeated_notes_from_time_signature](#)
[jad.demos.ferneyhough.make_lilypond_file\(\)](#), [1000](#)
[1307](#) [make_repeated_notes_from_time_signatures\(\)](#)
[make_mozart_lilypond_file\(\)](#) (in module [ab-](#) (in module [ab-](#)
[jad.demos.mozart.make_mozart_lilypond_file\(\)](#), [jad.tools.scoretools.make_repeated_notes_from_time_signature](#)

1000 MakeNewFunctionTemplateScript (class in ab-
 make_repeated_notes_with_shorter_notes_at_end() jad.tools.developerscripttools.MakeNewFunctionTemplateScript
 (in module ab- 1357
 jad.tools.scoretools.make_repeated_notes_with_shorter_notes_at_end),
 1000 (in module ab-
 make_repeated_rests_from_time_signatures() jad.tools.sequencetools.map_sequence_elements_to_canonic_tu
 (in module ab- 1053
 jad.tools.scoretools.make_repeated_rests_from_time_signatures), elements_to_numbered_sublists()
 1001 (in module ab-
 make_repeated_skips_from_time_signatures() jad.tools.sequencetools.map_sequence_elements_to_numbered
 (in module ab- 1054
 jad.tools.scoretools.make_repeated_skips_from_time_signatures), (class in ab-
 1001 jad.tools.instrumenttools.Marimba), 285
 make_rests() (in module ab- Markup (class in abjad.tools.markuptools.Markup), 413
 jad.tools.scoretools.make_rests), 1001 MarkupCommand (class in ab-
 make_rhythmic_sketch_staff() (in module ab- jad.tools.markuptools.MarkupCommand),
 jad.tools.scoretools.make_rhythmic_sketch_staff), 415
 1001 MarkupInventory (class in ab-
 make_row_of_nested_tuplets() (in module ab- jad.tools.markuptools.MarkupInventory),
 jad.demos.ferneyhough.make_row_of_nested_tuplets), 417
 1307 Matrix (class in abjad.tools.datastructuretools.Matrix),
 make_rows_of_nested_tuplets() (in module ab- 59
 jad.demos.ferneyhough.make_rows_of_nested_tuplets), Measure (class in abjad.tools.scoretools.Measure), 906
 1307 MeasuredComplexBeam (class in ab-
 make_score() (in module ab- jad.tools.spannertools.MeasuredComplexBeam),
 jad.demos.ferneyhough.make_score), 1307 1125
 make_skips_with_multiplied_durations() MeasurewiseAttackPointOptimizer (class in ab-
 (in module ab- jad.tools.quantizationtools.MeasurewiseAttackPointOptimizer),
 jad.tools.scoretools.make_skips_with_multiplied_durations), 573
 1002 MeasurewiseQSchema (class in ab-
 make_solid_text_spanner_with_nib() (in module ab- jad.tools.quantizationtools.MeasurewiseQSchema),
 jad.tools.spannertools.make_solid_text_spanner_with_nib), 74
 1164 MeasurewiseQSchemaItem (class in ab-
 make_spacer_skip_measures() (in module ab- jad.tools.quantizationtools.MeasurewiseQSchemaItem),
 jad.tools.scoretools.make_spacer_skip_measures), 678
 1002 MeasurewiseQTarget (class in ab-
 make_spacing_vector() (in module ab- jad.tools.quantizationtools.MeasurewiseQTarget),
 jad.tools.layouttools.make_spacing_vector), 680
 374 merge_duration_sequences() (in module ab-
 make_test_time_segments() (in module ab- jad.tools.sequencetools.merge_duration_sequences),
 jad.tools.quantizationtools.make_test_time_segments), 1054
 737 Meter (class in abjad.tools.metertools.Meter), 459
 make_text_alignment_example_lilypond_file() MetricAccentKernel (class in ab-
 (in module ab- jad.tools.metertools.MetricAccentKernel),
 jad.tools.documentationtools.make_text_alignment_example_lilypond_file),
 1517 MezzoSopranoVoice (class in ab-
 make_tied_leaf() (in module ab- jad.tools.instrumenttools.MezzoSopranoVoice),
 jad.tools.scoretools.make_tied_leaf), 1002 288
 make_time_signature_context_block() (in module ab- MIDIBlock (class in ab-
 jad.tools.lilypondfiletools.make_time_signature_context_block), jad.tools.lilypondfiletools.MIDIBlock),
 411 403
 make_vertically_adjusted_composer_markup() MissingMeasureError (class in ab-
 (in module ab- jad.tools.exceptiontools.MissingMeasureError),
 jad.tools.markuptools.make_vertically_adjusted_composer_markup), 503
 424 MissingSpannerError (class in ab-
 MakeNewClassTemplateScript (class in ab- jad.tools.exceptiontools.MissingSpannerError),
 jad.tools.developerscripttools.MakeNewClassTemplateScript), 1504
 1355 MissingTempoError (class in ab-

[jad.tools.exceptiontools.MissingTempoError](#)),
[1525](#)
 Mode (class in [abjad.tools.tonalanalysis tools.Mode](#)),
[1272](#)
 ModuleCrawler (class in [abjad.tools.documentation tools.ModuleCrawler](#)),
[1425](#)
 move_full_measure_tuplet_prolation_to_measure_time_signature() ([in module abjad.tools.sequencetools.negative_absolute_value_of_sequence_elements_at_indices](#)),
[1054](#)
 negate_sequence_elements_at_indices() ([in module abjad.tools.sequencetools.negative_absolute_value_of_sequence_elements_at_indices](#)),
[1055](#)
 negate_sequence_elements_cyclically() ([in module abjad.tools.sequencetools.negative_absolute_value_of_sequence_elements_at_indices](#)),
[1055](#)
 move_full_measure_tuplet_prolation_to_measure_time_signature() ([in module abjad.tools.sequencetools.negative_absolute_value_of_sequence_elements_at_indices](#)),
[1003](#)
 move_measure_prolation_to_full_measure_tuplet() ([in module abjad.tools.math tools.NegativeInfinity](#)),
[428](#)
 new() ([in module abjad.tools.toplevel tools.new](#)),
[1298](#)
 next_integer_partition() ([in module abjad.tools.math tools.next_integer_partition](#)),
[453](#)
 MultimeasureRest (class in [abjad.tools.score tools.MultimeasureRest](#)),
[915](#)
 NonattributedBlock (class in [abjad.tools.lilypond file tools.NonattributedBlock](#)),
[380](#)
 MultipartBeam (class in [abjad.tools.spanner tools.MultipartBeam](#)),
[1130](#)
 NonreducedFraction (class in [abjad.tools.math tools.NonreducedFraction](#)),
[430](#)
 Multiplier (class in [abjad.tools.duration tools.Multiplier](#)),
[123](#)
 NonreducedRatio (class in [abjad.tools.math tools.NonreducedRatio](#)),
[438](#)
 Music (class in [abjad.tools.lilypond parser tools.Music](#)),
[1537](#)
 Note (class in [abjad.tools.score tools.Note](#)),
[917](#)
 MusicGlyph (class in [abjad.tools.markup tools.MusicGlyph](#)),
[421](#)
 NoteHead (class in [abjad.tools.score tools.NoteHead](#)),
[919](#)
 mutate() ([in module abjad.tools.toplevel tools.mutate](#)),
[1298](#)
 NoteHeadInventory (class in [abjad.tools.score tools.NoteHeadInventory](#)),
[922](#)
 MutationAgent (class in [abjad.tools.agent tools.MutationAgent](#)),
[18](#)
 NoteRhythmMaker (class in [abjad.tools.rhythm maker tools.NoteRhythmMaker](#)),
[771](#)
N
 NaiveAttackPointOptimizer (class in [abjad.tools.quantization tools.NaiveAttackPointOptimizer](#)),
[682](#)
 notes_and_chords_are_in_range() ([in module abjad.tools.instrument tools.notes_and_chords_are_in_range](#)),
[354](#)
 named_pitch_and_clef_to_staff_position_number() ([in module abjad.tools.instrument tools.notes_and_chords_are_in_range](#)),
[354](#)
 NamedInterval (class in [abjad.tools.pitch tools.NamedInterval](#)),
[539](#)
 NullAttackPointOptimizer (class in [abjad.tools.quantization tools.NullAttackPointOptimizer](#)),
[683](#)
 NamedIntervalClass (class in [abjad.tools.pitch tools.NamedIntervalClass](#)),
[544](#)
 numbered_inversion_equivalent_interval_class_dictionary() ([in module abjad.tools.pitch tools.numbered_inversion_equivalent_interval_class_dictionary](#)),
[641](#)
 NamedInversionEquivalentIntervalClass (class in [abjad.tools.pitch tools.NamedInversionEquivalentIntervalClass](#)),
[546](#)
 NumberedInterval (class in [abjad.tools.pitch tools.NumberedInterval](#)),
[560](#)
 NamedPitch (class in [abjad.tools.pitch tools.NamedPitch](#)),
[548](#)
 NumberedIntervalClass (class in [abjad.tools.pitch tools.NumberedIntervalClass](#)),
[563](#)
 NamedPitchClass (class in [abjad.tools.pitch tools.NamedPitchClass](#)),
[555](#)
 NumberedInversionEquivalentIntervalClass (class in [abjad.tools.pitch tools.NumberedInversionEquivalentIntervalClass](#)),
[565](#)
 negate_absolute_value_of_sequence_elements_at_indices() ([in module abjad.tools.sequencetools.negative_absolute_value_of_sequence_elements_at_indices](#)),
[1054](#)
 NumberedPitch (class in [ab-](#)

jad.tools.pitchtools.NumberedPitch), 567

NumberedPitchClass (class in abjad.tools.pitchtools.NumberedPitchClass), 573

NumberedPitchClassColorMap (class in abjad.tools.pitchtools.NumberedPitchClassColorMap), 578

O

Oboe (class in abjad.tools.instrumenttools.Oboe), 291

OctavationSpanner (class in abjad.tools.spannertools.OctavationSpanner), 1133

Octave (class in abjad.tools.pitchtools.Octave), 580

OctaveTranspositionMapping (class in abjad.tools.pitchtools.OctaveTranspositionMapping), 583

OctaveTranspositionMappingComponent (class in abjad.tools.pitchtools.OctaveTranspositionMappingComponent), 588

OctaveTranspositionMappingInventory (class in abjad.tools.pitchtools.OctaveTranspositionMappingInventory), 589

Offset (class in abjad.tools.durationtools.Offset), 134

offset_happens_after_timespan_starts() (in module abjad.tools.timespantools.offset_happens_after_timespan_starts), 1251

offset_happens_after_timespan_stops() (in module abjad.tools.timespantools.offset_happens_after_timespan_stops), 1251

offset_happens_before_timespan_starts() (in module abjad.tools.timespantools.offset_happens_before_timespan_starts), 1252

offset_happens_before_timespan_stops() (in module abjad.tools.timespantools.offset_happens_before_timespan_stops), 1253

offset_happens_during_timespan() (in module abjad.tools.timespantools.offset_happens_during_timespan), 1253

offset_happens_when_timespan_starts() (in module abjad.tools.timespantools.offset_happens_when_timespan_starts), 1254

offset_happens_when_timespan_stops() (in module abjad.tools.timespantools.offset_happens_when_timespan_stops), 1254

OffsetTimespanTimeRelation (class in abjad.tools.timespantools.OffsetTimespanTimeRelation), 1192

OrdinalConstant (class in abjad.tools.datastructuretools.OrdinalConstant), 61

OutputBurnishedTaleaRhythmMaker (class in abjad.tools.rhythmmakertools.OutputBurnishedTaleaRhythmMaker), 774

OutputFormat (class in abjad.tools.abjadbooktools.OutputFormat), 1321

OutputIncisedNoteRhythmMaker (class in abjad.tools.rhythmmakertools.OutputIncisedNoteRhythmMaker), 778

OutputIncisedRestRhythmMaker (class in abjad.tools.rhythmmakertools.OutputIncisedRestRhythmMaker), 781

OutputIncisedRhythmMaker (class in abjad.tools.rhythmmakertools.OutputIncisedRhythmMaker), 747

OverfullContainerError (class in abjad.tools.exceptiontools.OverfullContainerError), 1526

override() (in module abjad.tools.topleveltools.override), 1299

overwrite_sequence_elements_at_indices() (in module abjad.tools.sequencetools.overwrite_sequence_elements_at_indices), 1065

P

pair_duration_sequence_elements_with_input_pair_values() (in module abjad.tools.sequencetools.pair_duration_sequence_elements_with_input_pair_values), 1055

PaperBlock (class in abjad.tools.lilypondfiletools.PaperBlock), 406

ParallelJobHandler (class in abjad.tools.quantizationtools.ParallelJobHandler), 684

ParallelJobHandlerWorker (class in abjad.tools.quantizationtools.ParallelJobHandlerWorker), 685

Parentage (class in abjad.tools.selectiontools.Parentage), 1018

parse() (in module abjad.tools.topleveltools.parse), 1299

parse_reduced_ly_syntax() (in module abjad.tools.lilypondparsertools.parse_reduced_ly_syntax), 1589

parse_rtm_syntax() (in module abjad.tools.rhythmtreetools.parse_rtm_syntax), 825

Parser (class in abjad.tools.abctools.Parser), 1318

PartCantusScoreTemplate (class in abjad.demos.part.PartCantusScoreTemplate), 1311

partition_integer_by_ratio() (in module abjad.tools.mathtools.partition_integer_by_ratio), 453

partition_integer_into_canonic_parts() (in module abjad.tools.mathtools.partition_integer_into_canonic_parts), 453

partition_integer_into_halves() (in module abjad.tools.mathtools.partition_integer_into_halves), 454

partition_integer_into_parts_less_than_double()

(in module ab-	jad.tools.instrumenttools.PerformerInventory),
jad.tools.mathtools.partition_integer_into_parts_less_than_300),	
455	permute_named_pitch_carrier_list_by_twelve_tone_row()
partition_integer_into_units() (in module ab-	(in module ab-
jad.tools.mathtools.partition_integer_into_units),	jad.tools.pitchtools.permute_named_pitch_carrier_list_by_twel
455	641
partition_sequence_by_backgrounded_weights()	permute_sequence() (in module ab-
(in module ab-	jad.tools.sequencetools.permute_sequence),
jad.tools.sequencetools.partition_sequence_by_backgrounded_weights),	1062
1056	persist() (in module abjad.tools.topleveltools.persist),
partition_sequence_by_counts() (in module ab-	1299
jad.tools.sequencetools.partition_sequence_by_counts),	PersistenceAgent (class in ab-
1056	jad.tools.agenttools.PersistenceAgent),
partition_sequence_by_ratio_of_lengths()	33
(in module ab-	PhrasingSlur (class in ab-
jad.tools.sequencetools.partition_sequence_by_ratio_of_lengths),	jad.tools.spannertools.PhrasingSlur), 1137
1058	Piano (class in abjad.tools.instrumenttools.Piano), 305
partition_sequence_by_ratio_of_weights()	PianoPedalSpanner (class in ab-
(in module ab-	jad.tools.spannertools.PianoPedalSpanner),
jad.tools.sequencetools.partition_sequence_by_ratio_of_weights),	
1058	PianoStaff (class in abjad.tools.scoretools.PianoStaff),
partition_sequence_by_restricted_growth_function()	927
(in module ab-	Piccolo (class in abjad.tools.instrumenttools.Piccolo),
jad.tools.sequencetools.partition_sequence_by_restricted_growth_function),	308
1059	Pipe (class in abjad.tools.documentationtools.Pipe),
partition_sequence_by_sign_of_elements()	1426
(in module ab-	Pitch (class in abjad.tools.pitchtools.Pitch), 496
jad.tools.sequencetools.partition_sequence_by_sign_of_elements),	(class in ab-
1059	jad.tools.pitcharraytools.PitchArray), 469
partition_sequence_by_value_of_elements()	PitchArrayCell (class in ab-
(in module ab-	jad.tools.pitcharraytools.PitchArrayCell),
jad.tools.sequencetools.partition_sequence_by_value_of_elements),	
1060	PitchArrayColumn (class in ab-
partition_sequence_by_weights_at_least()	jad.tools.pitcharraytools.PitchArrayColumn),
(in module ab-	478
jad.tools.sequencetools.partition_sequence_by_weights_at_least),	PitchArrayInventory (class in ab-
1060	jad.tools.pitcharraytools.PitchArrayInventory),
partition_sequence_by_weights_at_most()	481
(in module ab-	PitchArrayRow (class in ab-
jad.tools.sequencetools.partition_sequence_by_weights_at_most),	jad.tools.pitcharraytools.PitchArrayRow),
1060	486
partition_sequence_by_weights_exactly()	PitchClass (class in abjad.tools.pitchtools.PitchClass),
(in module ab-	500
jad.tools.sequencetools.partition_sequence_by_weights_exactly),	PitchClassSegment (class in ab-
1061	jad.tools.pitchtools.PitchClassSegment),
partition_sequence_extended_to_counts()	594
(in module ab-	PitchClassSet (class in ab-
jad.tools.sequencetools.partition_sequence_extended_to_counts),	jad.tools.pitchtools.PitchClassSet), 599
1062	PitchClassVector (class in ab-
PartitionError (class in ab-	jad.tools.pitchtools.PitchClassVector),
jad.tools.exceptiontools.PartitionError),	604
1527	PitchedQEvent (class in ab-
PayloadTree (class in ab-	jad.tools.quantizationtools.PitchedQEvent),
jad.tools.datastructuretools.PayloadTree),	687
63	PitchRange (class in ab-
Performer (class in ab-	jad.tools.pitchtools.PitchRange), 607
jad.tools.instrumenttools.Performer), 294	PitchRangeInventory (class in ab-
PerformerInventory (class in ab-	jad.tools.pitchtools.PitchRangeInventory),

611
PitchSegment (class in abjad.tools.pitchtools.PitchSegment), 615
PitchSet (class in abjad.tools.pitchtools.PitchSet), 621
PitchVector (class in abjad.tools.pitchtools.PitchVector), 625
play() (in module abjad.tools.topleveltools.play), 1299
pluralize_string() (in module abjad.tools.stringtools.pluralize_string), 1173
PyTestScript (class in abjad.tools.developerscripttools.PyTestScript), 1360

Q

QEvent (class in abjad.tools.quantizationtools.QEvent), 653
QEventProxy (class in abjad.tools.quantizationtools.QEventProxy), 688
QEventSequence (class in abjad.tools.quantizationtools.QEventSequence), 690
QGrid (class in abjad.tools.quantizationtools.QGrid), 696
QGridContainer (class in abjad.tools.quantizationtools.QGridContainer), 699
QGridLeaf (class in abjad.tools.quantizationtools.QGridLeaf), 711
QSchema (class in abjad.tools.quantizationtools.QSchema), 654
QSchemaItem (class in abjad.tools.quantizationtools.QSchemaItem), 656
QTarget (class in abjad.tools.quantizationtools.QTarget), 657
QTargetBeat (class in abjad.tools.quantizationtools.QTargetBeat), 717
QTargetMeasure (class in abjad.tools.quantizationtools.QTargetMeasure), 719
QuantizationJob (class in abjad.tools.quantizationtools.QuantizationJob), 723
Quantizer (class in abjad.tools.quantizationtools.Quantizer), 725

R

Ratio (class in abjad.tools.mathtools.Ratio), 440
RedirectedStreams (class in abjad.tools.systemtools.RedirectedStreams), 1609
ReducedLyParser (class in abjad.tools.lilypondparsertools.ReducedLyParser),

1577
register_pitch_class_numbers_by_pitch_number_aggregate() (in module abjad.tools.pitchtools.register_pitch_class_numbers_by_pitch_number_aggregate), 642
remap_sequence_by_range_pairs() (in module abjad.tools.sequencetools.remap_sequence_by_range_pairs), 1062
remove_markup_from_leaves_in_expr() (in module abjad.tools.labeltools.remove_markup_from_leaves_in_expr), 371
remove_powers_of_two() (in module abjad.tools.mathtools.remove_powers_of_two), 456
remove_sequence_elements_at_indices() (in module abjad.tools.sequencetools.remove_sequence_elements_at_indices), 1063
remove_sequence_elements_at_indices_cyclically() (in module abjad.tools.sequencetools.remove_sequence_elements_at_indices_cyclically), 1063
remove_subsequence_of_weight_at_index() (in module abjad.tools.sequencetools.remove_subsequence_of_weight_at_index), 1063
RenameModulesScript (class in abjad.tools.developerscripttools.RenameModulesScript), 1362
repeat_runs_in_sequence_to_count() (in module abjad.tools.sequencetools.repeat_runs_in_sequence_to_count), 1063
repeat_sequence_elements_at_indices() (in module abjad.tools.sequencetools.repeat_sequence_elements_at_indices), 1064
repeat_sequence_elements_at_indices_cyclically() (in module abjad.tools.sequencetools.repeat_sequence_elements_at_indices_cyclically), 1064
repeat_sequence_elements_n_times_each() (in module abjad.tools.sequencetools.repeat_sequence_elements_n_times_each), 1065
repeat_sequence_n_times() (in module abjad.tools.sequencetools.repeat_sequence_n_times), 1065
repeat_sequence_to_length() (in module abjad.tools.sequencetools.repeat_sequence_to_length), 1065
repeat_sequence_to_weight_at_least() (in module abjad.tools.sequencetools.repeat_sequence_to_weight_at_least), 1065
repeat_sequence_to_weight_at_most() (in module abjad.tools.sequencetools.repeat_sequence_to_weight_at_most), 1066
repeat_sequence_to_weight_exactly() (in module abjad.tools.sequencetools.repeat_sequence_to_weight_exactly), 1066

replace_contents_of_measures_in_expr()	ReSTTOCItem	(class in ab-
(in module ab-	jad.tools.documentationtools.ReSTTOCItem),	
jad.tools.scoretools.replace_contents_of_measures_in_expr(),	1003	
retain_sequence_elements_at_indices()	(in module ab-	
replace_sequence_elements_cyclically_with_new_material()	jad.tools.sequencetools.retain_sequence_elements_at_indices),	
(in module ab-	1066	
jad.tools.sequencetools.replace_sequence_elements_cyclically_with_new_material()	retain_sequence_elements_at_indices_cyclically()	
1066	(in module ab-	
ReplaceInFilesScript	(class in ab-	
jad.tools.developerscripttools.ReplaceInFilesScript),	1067	
1365	reverse_sequence()	(in module ab-
requires() (in module abjad.tools.systemtools.requires),	jad.tools.sequencetools.reverse_sequence),	
1620	1067	
ResidueClass	(class in ab-	
jad.tools.sievetools.ResidueClass),	1079	
Rest (class in abjad.tools.scoretools.Rest),	934	
ReSTAutodocDirective	(class in ab-	
jad.tools.documentationtools.ReSTAutodocDirective),	1428	
ReSTAutosummaryDirective	(class in ab-	
jad.tools.documentationtools.ReSTAutosummaryDirective),	1438	
ReSTAutosummaryItem	(class in ab-	
jad.tools.documentationtools.ReSTAutosummaryItem),	1447	
ReSTDDirective	(class in ab-	
jad.tools.documentationtools.ReSTDDirective),	1375	
ReSTDDocument	(class in ab-	
jad.tools.documentationtools.ReSTDDocument),	1451	
ReSTHeading	(class in ab-	
jad.tools.documentationtools.ReSTHeading),	1461	
ReSTHorizontalRule	(class in ab-	
jad.tools.documentationtools.ReSTHorizontalRule),	1465	
ReSTInheritanceDiagram	(class in ab-	
jad.tools.documentationtools.ReSTInheritanceDiagram),	1469	
ReSTLineageDirective	(class in ab-	
jad.tools.documentationtools.ReSTLineageDirective),	1478	
ReSTOnlyDirective	(class in ab-	
jad.tools.documentationtools.ReSTOnlyDirective),	1488	
ReSTOutputFormat	(class in ab-	
jad.tools.abjadbooktools.ReSTOutputFormat),	1332	
ReSTParagraph	(class in ab-	
jad.tools.documentationtools.ReSTParagraph),	1497	
RestRhythmMaker	(class in ab-	
jad.tools.rhythmmakertools.RestRhythmMaker)	785	
ReSTTOCDirective	(class in ab-	
jad.tools.documentationtools.ReSTTOCDirective),	1501	
	reverse_sequence_elements()	(in module ab-
	jad.tools.sequencetools.reverse_sequence_elements),	
	1067	
	RhythmicStaff	(class in ab-
	jad.tools.scoretools.RhythmicStaff),	936
	RhythmMaker	(class in ab-
	jad.tools.rhythmmakertools.RhythmMaker),	749
	RhythmTreeContainer	(class in ab-
	jad.tools.rhythmtreetools.RhythmTreeContainer),	803
	RhythmTreeLeaf	(class in ab-
	jad.tools.rhythmtreetools.RhythmTreeLeaf),	816
	RhythmTreeNode	(class in ab-
	jad.tools.rhythmtreetools.RhythmTreeNode),	797
	RhythmTreeParser	(class in ab-
	jad.tools.rhythmtreetools.RhythmTreeParser),	822
	RomanNumeral	(class in ab-
	jad.tools.tonalanalysisistools.RomanNumeral),	1273
	RootedChordClass	(class in ab-
	jad.tools.tonalanalysisistools.RootedChordClass),	1275
	RootlessChordClass	(class in ab-
	jad.tools.tonalanalysisistools.RootlessChordClass),	1281
	rotate_sequence()	(in module ab-
	jad.tools.sequencetools.rotate_sequence),	1067
	run_abjad()	(in module ab-
	jad.tools.systemtools.run_abjad),	1620
	run_abjadbook()	(in module ab-
	jad.tools.developerscripttools.run_abjadbook),	1372
	run_ajv()	(in module ab-
	jad.tools.developerscripttools.run_ajv),	1372
	RunDoctestsScript	(class in ab-
	jad.tools.developerscripttools.RunDoctestsScript),	1368

S

Scale (class in abjad.tools.tonalanalysistools.Scale), 1286

scale_measure_denominator_and_adjust_measure_contents() (in module abjad.tools.scoretools.scale_measure_denominator_and_adjust_measure_contents), 1004

ScaleDegree (class in abjad.tools.tonalanalysistools.ScaleDegree), 1291

Scheme (class in abjad.tools.schemetools.Scheme), 827

SchemeAssociativeList (class in abjad.tools.schemetools.SchemeAssociativeList), 830

SchemeColor (class in abjad.tools.schemetools.SchemeColor), 832

SchemeMoment (class in abjad.tools.schemetools.SchemeMoment), 834

SchemePair (class in abjad.tools.schemetools.SchemePair), 836

SchemeParser (class in abjad.tools.lilypondparsertools.SchemeParser), 1583

SchemeParserFinishedError (class in abjad.tools.exceptiontools.SchemeParserFinishedError), 1528

SchemeVector (class in abjad.tools.schemetools.SchemeVector), 838

SchemeVectorConstant (class in abjad.tools.schemetools.SchemeVectorConstant), 840

Score (class in abjad.tools.scoretools.Score), 943

ScoreBlock (class in abjad.tools.lilypondfiletools.ScoreBlock), 408

SearchTree (class in abjad.tools.quantizationtools.SearchTree), 659

Segment (class in abjad.tools.pitchtools.Segment), 503

select() (in module abjad.tools.tonalanalysistools.select), 1296

select() (in module abjad.tools.topleveltools.select), 1300

Selection (class in abjad.tools.selectiontools.Selection), 1022

SelectionInventory (class in abjad.tools.selectiontools.SelectionInventory), 1024

SequentialMusic (class in abjad.tools.lilypondparsertools.SequentialMusic), 1587

SerialJobHandler (class in abjad.tools.quantizationtools.SerialJobHandler), 729

Set (class in abjad.tools.pitchtools.Set), 506

set_always_format_time_signature_of_measures_in_expr() (in module abjad.tools.scoretools.set_always_format_time_signature_of_measures_in_expr), 1004

set_line_breaks_by_line_duration() (in module abjad.tools.layouttools.set_line_breaks_by_line_duration), 375

set_line_breaks_cyclically_by_line_duration_ge() (in module abjad.tools.layouttools.set_line_breaks_cyclically_by_line_duration_ge), 375

set_line_breaks_cyclically_by_line_duration_in_seconds_ge() (in module abjad.tools.layouttools.set_line_breaks_cyclically_by_line_duration_in_seconds_ge), 376

set_measure_denominator_and_adjust_numerator() (in module abjad.tools.scoretools.set_measure_denominator_and_adjust_numerator), 1005

set_written_pitch_of_pitched_components_in_expr() (in module abjad.tools.pitchtools.set_written_pitch_of_pitched_components_in_expr), 642

shadow_pitch_contour_reservoir() (in module abjad.demos.part.shadow_pitch_contour_reservoir), 1314

show() (in module abjad.tools.topleveltools.show), 1300

Sieve (class in abjad.tools.sievetools.Sieve), 1081

sign() (in module abjad.tools.mathtools.sign), 456

SilentQEvent (class in abjad.tools.quantizationtools.SilentQEvent), 730

SimpleInequality (class in abjad.tools.timespantools.SimpleInequality), 1194

SimultaneousMusic (class in abjad.tools.lilypondparsertools.SimultaneousMusic), 1539

SimultaneousSelection (class in abjad.tools.selectiontools.SimultaneousSelection), 1028

Skip (class in abjad.tools.scoretools.Skip), 950

SkipRhythmMaker (class in abjad.tools.rhythmmakertools.SkipRhythmMaker), 788

SliceSelection (class in abjad.tools.selectiontools.SliceSelection), 1030

Slur (class in abjad.tools.spannertools.Slur), 1143

snake_case_to_lower_camel_case() (in module abjad.tools.stringtools.snake_case_to_lower_camel_case), 1173

snake_case_to_upper_camel_case() (in module abjad.tools.stringtools.snake_case_to_upper_camel_case), 1173

SopraninoSaxophone (class in abjad.tools.instrumenttools.SopraninoSaxophone), 311

SopranoSaxophone	(class in abjad.tools.instrumenttools.SopranoSaxophone),	jad.tools.systemtools.StorageFormatSpecification),
314		1612
SopranoVoice	(class in abjad.tools.instrumenttools.SopranoVoice),	string_to_accent_free_snake_case() (in module ab-
317		jad.tools.stringtools.string_to_accent_free_snake_case),
sort_named_pitch_carriers_in_expr() (in module ab-		1173
jad.tools.pitchtools.sort_named_pitch_carriers_in_expr),		string_to_space_delimited_lowercase() (in module ab-
642		jad.tools.stringtools.string_to_space_delimited_lowercase),
SortedCollection	(class in abjad.tools.datastructuretools.SortedCollection),	StringOrchestraScoreTemplate (class in ab-
76		jad.tools.templatetools.StringOrchestraScoreTemplate),
space_delimited_lowercase_to_upper_camel_case()		1181
(in module ab-		StringQuartetScoreTemplate (class in ab-
jad.tools.stringtools.space_delimited_lowercase_to_upper_camel_case),		jad.tools.templatetools.StringQuartetScoreTemplate),
1173		1182
SpacingIndication	(class in abjad.tools.layouttools.SpacingIndication),	strip_diacritics_from_binary_string() (in module ab-
373		jad.tools.stringtools.strip_diacritics_from_binary_string),
Spanner	(class in abjad.tools.spannertools.Spanner),	1174
1146		suggest_clef_for_named_pitches() (in module ab-
spell_numbered_interval_number() (in module ab-		jad.tools.pitchtools.suggest_clef_for_named_pitches),
jad.tools.pitchtools.spell_numbered_interval_number),		643
642		sum_consecutive_sequence_elements_by_sign()
spell_pitch_number() (in module ab-		(in module ab-
jad.tools.pitchtools.spell_pitch_number),		jad.tools.sequencetools.sum_consecutive_sequence_elements_by_sign),
642		1069
splice_new_elements_between_sequence_elements()		sum_sequence_elements_at_indices() (in module ab-
(in module ab-		jad.tools.sequencetools.sum_sequence_elements_at_indices),
jad.tools.sequencetools.splice_new_elements_between_sequence_elements),		1070
1067		SyntaxNode (class in ab-
split_sequence_by_weights() (in module ab-		jad.tools.lilypondparsertools.SyntaxNode),
jad.tools.sequencetools.split_sequence_by_weights),		458
1068		T
split_sequence_extended_to_weights() (in module ab-		TaleaRhythmMaker (class in ab-
jad.tools.sequencetools.split_sequence_extended_to_weights),		jad.tools.rhythmmakertools.TaleaRhythmMaker),
1069		790
Staff	(class in abjad.tools.scoretools.Staff),	Tempo (class in abjad.tools.indicatortools.Tempo),
952		174
StaffChange	(class in abjad.tools.indicatortools.StaffChange),	TempoInventory (class in ab-
171		jad.tools.indicatortools.TempoInventory),
StaffGroup	(class in abjad.tools.scoretools.StaffGroup),	179
959		TenorSaxophone (class in ab-
StaffLinesSpanner	(class in abjad.tools.spannertools.StaffLinesSpanner),	jad.tools.instrumenttools.TenorSaxophone),
1149		320
StatalServer	(class in abjad.tools.datastructuretools.StatalServer),	TenorTrombone (class in ab-
79		jad.tools.instrumenttools.TenorTrombone),
StatalServerCursor	(class in abjad.tools.datastructuretools.StatalServerCursor),	323
80		TenorVoice (class in ab-
StemTremolo	(class in abjad.tools.indicatortools.StemTremolo),	jad.tools.instrumenttools.TenorVoice),
172		326
StorageFormatManager	(class in abjad.tools.systemtools.StorageFormatManager),	TerminalQEvent (class in ab-
1610		jad.tools.quantizationtools.TerminalQEvent),
StorageFormatSpecification	(class in ab-	731
		TestAndRebuildScript (class in ab-
		jad.tools.developerscripttools.TestAndRebuildScript),
		1370
		TestManager (class in ab-
		jad.tools.systemtools.TestManager),
		1614
		TextScriptSpanner (class in ab-
		jad.tools.spannertools.TextScriptSpanner),

1152				timespan_2_starts_after_timespan_1_stops()	
TextSpanner	(class in ab-			(in module ab-	
	jad.tools.spannertools.TextSpanner),	1155		jad.tools.timespantools.timespan_2_starts_after_timespan_1_stops()	
Tie	(class in abjad.tools.spannertools.Tie),	1158		1259	
Timer	(class in abjad.tools.systemtools.Timer),	1615		timespan_2_starts_before_timespan_1_starts()	
TimeRelation	(class in ab-			(in module ab-	
	jad.tools.timespantools.TimeRelation),			jad.tools.timespantools.timespan_2_starts_before_timespan_1_starts()	
	1185			1259	
TimeSignature	(class in ab-			timespan_2_starts_before_timespan_1_stops()	
	jad.tools.indicatortools.TimeSignature),			(in module ab-	
	183			jad.tools.timespantools.timespan_2_starts_before_timespan_1_stops()	
Timespan	(class in ab-			1259	
	jad.tools.timespantools.Timespan),	1196		timespan_2_starts_during_timespan_1() (in module ab-	
timespan_2_contains_timespan_1_improperly()				jad.tools.timespantools.timespan_2_starts_during_timespan_1()),	
	(in module ab-			1260	
	jad.tools.timespantools.timespan_2_contains_timespan_1_improperly()			timespan_2_starts_when_timespan_1_starts()	
	1254			(in module ab-	
timespan_2_curtails_timespan_1() (in module ab-				jad.tools.timespantools.timespan_2_starts_when_timespan_1_stops())	
	jad.tools.timespantools.timespan_2_curtails_timespan_1()),	1261		timespan_2_starts_when_timespan_1_stops()	
	1255			(in module ab-	
timespan_2_delays_timespan_1() (in module ab-				jad.tools.timespantools.timespan_2_starts_when_timespan_1_stops())	
	jad.tools.timespantools.timespan_2_delays_timespan_1()),			1261	
	1255			timespan_2_stops_after_timespan_1_starts()	
timespan_2_happens_during_timespan_1()				(in module ab-	
	(in module ab-			jad.tools.timespantools.timespan_2_stops_after_timespan_1_starts())	
	jad.tools.timespantools.timespan_2_happens_during_timespan_1()),	1261		timespan_2_stops_after_timespan_1_stops()	
	1256			(in module ab-	
timespan_2_intersects_timespan_1() (in module ab-				jad.tools.timespantools.timespan_2_stops_after_timespan_1_stops())	
	jad.tools.timespantools.timespan_2_intersects_timespan_1()),			1262	
	1256			timespan_2_stops_before_timespan_1_starts()	
timespan_2_is_congruent_to_timespan_1()				(in module ab-	
	(in module ab-			jad.tools.timespantools.timespan_2_stops_before_timespan_1_starts())	
	jad.tools.timespantools.timespan_2_is_congruent_to_timespan_1()),			1262	
	1256			timespan_2_stops_before_timespan_1_stops()	
timespan_2_overlaps_all_of_timespan_1()				(in module ab-	
	(in module ab-			jad.tools.timespantools.timespan_2_stops_before_timespan_1_stops())	
	jad.tools.timespantools.timespan_2_overlaps_all_of_timespan_1()),			1262	
	1257			timespan_2_stops_during_timespan_1() (in module ab-	
timespan_2_overlaps_only_start_of_timespan_1()				jad.tools.timespantools.timespan_2_stops_during_timespan_1()),	
	(in module ab-			1263	
	jad.tools.timespantools.timespan_2_overlaps_only_start_of_timespan_1()),			timespan_2_stops_when_timespan_1_starts()	
	1257			(in module ab-	
timespan_2_overlaps_only_stop_of_timespan_1()				jad.tools.timespantools.timespan_2_stops_when_timespan_1_stops())	
	(in module ab-			1263	
	jad.tools.timespantools.timespan_2_overlaps_only_stop_of_timespan_1()),			timespan_2_stops_when_timespan_1_stops()	
	1257			(in module ab-	
timespan_2_overlaps_start_of_timespan_1()				jad.tools.timespantools.timespan_2_stops_when_timespan_1_stops())	
	(in module ab-			1263	
	jad.tools.timespantools.timespan_2_overlaps_start_of_timespan_1()),			timespan_2_trisects_timespan_1() (in module ab-	
	1258			jad.tools.timespantools.timespan_2_trisects_timespan_1()),	
timespan_2_overlaps_stop_of_timespan_1()				1258	
	(in module ab-			TimespanInventory	
	jad.tools.timespantools.timespan_2_overlaps_stop_of_timespan_1()),			(class in ab-	
	1258			jad.tools.timespantools.TimespanInventory),	
timespan_2_starts_after_timespan_1_starts()				1216	
	(in module ab-			TimespanTimeRelation	
	jad.tools.timespantools.timespan_2_starts_after_timespan_1_starts()),			(class in ab-	
	1258			jad.tools.timespantools.TimespanTimeRelation),	

1246
TonalAnalysisAgent (class in abjad.tools.pitchtools.TwelveToneRow),
jad.tools.tonalanalysistools.TonalAnalysisAgent), 629
1293
ToolsPackageDocumenter (class in abjad.tools.templatetools.TwoStaffPianoScoreTemplate),
jad.tools.documentationtools.ToolsPackageDocumenter), 1183
1514
transpose_from_sounding_pitch_to_written_pitch() jad.tools.datastructuretools.TypedCollection),
(in module abjad.tools.instrumenttools.transpose_from_sounding_pitch_to_written_pitch()), 35
355
transpose_from_written_pitch_to_sounding_pitch() jad.tools.datastructuretools.TypedCounter),
(in module abjad.tools.instrumenttools.transpose_from_written_pitch_to_sounding_pitch()), 95
TypedFrozenSet (class in abjad.tools.datastructuretools.TypedFrozenSet),
355
transpose_named_pitch_by_numbered_interval_and_respell() TypedList (class in abjad.tools.datastructuretools.TypedList),
(in module abjad.tools.pitchtools.transpose_named_pitch_by_numbered_interval_and_respell), 643
TypedTuple (class in abjad.tools.datastructuretools.TypedTuple),
transpose_pitch_carrier_by_interval() (in module abjad.tools.pitchtools.transpose_pitch_carrier_by_interval), 107
643
transpose_pitch_class_number_to_pitch_number_neighbors() UnboundedTimeIntervalError (class in abjad.tools.pitchtools.transpose_pitch_class_number_to_pitch_number_neighbors),
(in module abjad.tools.pitchtools.transpose_pitch_class_number_to_pitch_number_neighbors), 1529
643
transpose_pitch_expr_into_pitch_range() UnderfullContainerError (class in abjad.tools.exceptiontools.UnderfullContainerError),
(in module abjad.tools.pitchtools.transpose_pitch_expr_into_pitch_range()), 330
644
transpose_pitch_number_by_octave_transposition_mapping() UntunedPercussion (class in abjad.tools.instrumenttools.UntunedPercussion),
(in module abjad.tools.pitchtools.transpose_pitch_number_by_octave_transposition_mapping()), 335
644
TreeContainer (class in abjad.tools.quantizationtools.UnweightedSearchTree),
jad.tools.datastructuretools.TreeContainer), 733
82
TreeNode (class in abjad.tools.systemtools.UpdateManager),
jad.tools.datastructuretools.TreeNode), 1617
91
TrillSpanner (class in abjad.tools.stringtools.upper_camel_case_to_snake_case),
jad.tools.spannertools.TrillSpanner), 1161
1174
Trumpet (class in abjad.tools.stringtools.upper_camel_case_to_space_delimited_lowercase()),
329
(in module abjad.tools.stringtools.upper_camel_case_to_space_delimited_lowercase()),
truncate_runs_in_sequence() (in module abjad.tools.stringtools.upper_camel_case_to_space_delimited_lowercase()),
jad.tools.sequencetools.truncate_runs_in_sequence(), 1174
1070
truncate_sequence_to_sum() (in module abjad.tools.pitchtools.Vector), 509
jad.tools.sequencetools.truncate_sequence_to_sum()), 1070
VerticalMoment (class in abjad.tools.selectiontools.VerticalMoment),
truncate_sequence_to_weight() (in module abjad.tools.selectiontools.VerticalMoment),
jad.tools.sequencetools.truncate_sequence_to_weight()), 1032
1071
Tuba (class in abjad.tools.instrumenttools.Vibraphone),
332
338
Tuplet (class in abjad.tools.instrumenttools.Viola), 341
966
TupletMonadRhythmMaker (class in abjad.tools.instrumenttools.Violin), 344
jad.tools.rhythmmakertools.TupletMonadRhythmMaker), 983
793

W

weight() (in module abjad.tools.mathtools.weight), [456](#)

WeightedSearchTree (class in abjad.tools.quantizationtools.WeightedSearchTree), [735](#)

WellformednessManager (class in abjad.tools.systemtools.WellformednessManager), [1618](#)

WoodwindFingering (class in abjad.tools.instrumenttools.WoodwindFingering), [347](#)

X

Xylophone (class in abjad.tools.instrumenttools.Xylophone), [351](#)

Y

yield_all_combinations_of_sequence_elements() (in module abjad.tools.sequencetools.yield_all_combinations_of_sequence_elements), [1071](#)

yield_all_compositions_of_integer() (in module abjad.tools.mathtools.yield_all_compositions_of_integer), [457](#)

yield_all_k_ary_sequences_of_length() (in module abjad.tools.sequencetools.yield_all_k_ary_sequences_of_length), [1072](#)

yield_all_pairs_between_sequences() (in module abjad.tools.sequencetools.yield_all_pairs_between_sequences), [1072](#)

yield_all_partitions_of_integer() (in module abjad.tools.mathtools.yield_all_partitions_of_integer), [457](#)

yield_all_partitions_of_sequence() (in module abjad.tools.sequencetools.yield_all_partitions_of_sequence), [1072](#)

yield_all_permutations_of_sequence() (in module abjad.tools.sequencetools.yield_all_permutations_of_sequence), [1073](#)

yield_all_permutations_of_sequence_in_orbit() (in module abjad.tools.sequencetools.yield_all_permutations_of_sequence_in_orbit), [1073](#)

yield_all_restricted_growth_functions_of_length() (in module abjad.tools.sequencetools.yield_all_restricted_growth_functions_of_length), [1073](#)

yield_all_rotations_of_sequence() (in module abjad.tools.sequencetools.yield_all_rotations_of_sequence), [1073](#)

yield_all_set_partitions_of_sequence() (in module abjad.tools.sequencetools.yield_all_set_partitions_of_sequence), [1074](#)

yield_all_subsequences_of_sequence() (in module abjad.tools.sequencetools.yield_all_subsequences_of_sequence), [1074](#)

yield_all_unordered_pairs_of_sequence()

(in module abjad.tools.sequencetools.yield_all_unordered_pairs_of_sequence), [1075](#)

yield_nonreduced_fractions() (in module abjad.tools.mathtools.yield_nonreduced_fractions), [457](#)

yield_outer_product_of_sequences() (in module abjad.tools.sequencetools.yield_outer_product_of_sequences), [1075](#)

Z

zip_sequences_cyclically() (in module abjad.tools.sequencetools.zip_sequences_cyclically), [1075](#)

zip_sequences_without_truncation() (in module abjad.tools.sequencetools.zip_sequences_without_truncation), [1076](#)