
Abjad API

Release 2.15

Trevor Bača, Josiah Wolf Oberholtzer, Víctor Adán

August 08, 2014

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	34
2	datastructuretools	39
2.1	Abstract classes	39
2.1.1	datastructuretools.TypedCollection	39
2.2	Concrete classes	41
2.2.1	datastructuretools.ContextMap	41
2.2.2	datastructuretools.CyclicMatrix	45
2.2.3	datastructuretools.CyclicPayloadTree	48
2.2.4	datastructuretools.CyclicTuple	63
2.2.5	datastructuretools.Matrix	65
2.2.6	datastructuretools.OrdinalConstant	68
2.2.7	datastructuretools.PayloadTree	70
2.2.8	datastructuretools.SortedCollection	84
2.2.9	datastructuretools.StatalServer	86
2.2.10	datastructuretools.StatalServerCursor	88
2.2.11	datastructuretools.TreeContainer	91
2.2.12	datastructuretools.TreeNode	100
2.2.13	datastructuretools.TypedCounter	104
2.2.14	datastructuretools.TypedFrozenSet	107
2.2.15	datastructuretools.TypedList	110
2.2.16	datastructuretools.TypedOrderedDict	115
2.2.17	datastructuretools.TypedTuple	118
3	durationtools	121
3.1	Concrete classes	121
3.1.1	durationtools.Duration	121
3.1.2	durationtools.Multiplier	133
3.1.3	durationtools.Offset	145
4	indicatortools	157
4.1	Concrete classes	157
4.1.1	indicatortools.Annotation	157
4.1.2	indicatortools.Arpeggio	159
4.1.3	indicatortools.Articulation	160
4.1.4	indicatortools.BarLine	163
4.1.5	indicatortools.BendAfter	164
4.1.6	indicatortools.BowContactPoint	166
4.1.7	indicatortools.BowPressure	168

4.1.8	indicatortools.Clef	169
4.1.9	indicatortools.ClefInventory	172
4.1.10	indicatortools.Dynamic	177
4.1.11	indicatortools.IndicatorExpression	179
4.1.12	indicatortools.IsAtSoundingPitch	181
4.1.13	indicatortools.IsUnpitched	182
4.1.14	indicatortools.KeyCluster	183
4.1.15	indicatortools.KeySignature	185
4.1.16	indicatortools.LaissezVibrer	187
4.1.17	indicatortools.LilyPondCommand	188
4.1.18	indicatortools.LilyPondComment	190
4.1.19	indicatortools.StaffChange	192
4.1.20	indicatortools.StemTremolo	194
4.1.21	indicatortools.StringContactPoint	196
4.1.22	indicatortools.StringNumber	197
4.1.23	indicatortools.Tempo	199
4.1.24	indicatortools.TempoInventory	204
4.1.25	indicatortools.TimeSignature	209
4.1.26	indicatortools.TimeSignatureInventory	213
4.1.27	indicatortools.Tuning	218
5	instrumenttools	223
5.1	Concrete classes	223
5.1.1	instrumenttools.Accordion	223
5.1.2	instrumenttools.AltoFlute	227
5.1.3	instrumenttools.AltoSaxophone	230
5.1.4	instrumenttools.AltoTrombone	233
5.1.5	instrumenttools.AltoVoice	236
5.1.6	instrumenttools.BaritoneSaxophone	239
5.1.7	instrumenttools.BaritoneVoice	242
5.1.8	instrumenttools.BassClarinet	245
5.1.9	instrumenttools.BassFlute	248
5.1.10	instrumenttools.BassSaxophone	251
5.1.11	instrumenttools.BassTrombone	254
5.1.12	instrumenttools.BassVoice	257
5.1.13	instrumenttools.Bassoon	260
5.1.14	instrumenttools.Cello	263
5.1.15	instrumenttools.ClarinetInA	266
5.1.16	instrumenttools.ClarinetInBFlat	269
5.1.17	instrumenttools.ClarinetInEFlat	272
5.1.18	instrumenttools.Contrabass	275
5.1.19	instrumenttools.ContrabassClarinet	278
5.1.20	instrumenttools.ContrabassFlute	281
5.1.21	instrumenttools.ContrabassSaxophone	284
5.1.22	instrumenttools.Contrabassoon	287
5.1.23	instrumenttools.EnglishHorn	290
5.1.24	instrumenttools.Flute	293
5.1.25	instrumenttools.FrenchHorn	296
5.1.26	instrumenttools.Glockenspiel	299
5.1.27	instrumenttools.Guitar	302
5.1.28	instrumenttools.Harp	305
5.1.29	instrumenttools.Harpsichord	308
5.1.30	instrumenttools.Instrument	311
5.1.31	instrumenttools.InstrumentInventory	313
5.1.32	instrumenttools.Marimba	318
5.1.33	instrumenttools.MezzoSopranoVoice	321
5.1.34	instrumenttools.Oboe	324
5.1.35	instrumenttools.Performer	327

5.1.36	instrumenttools.PerformerInventory	334
5.1.37	instrumenttools.Piano	339
5.1.38	instrumenttools.Piccolo	342
5.1.39	instrumenttools.SopraninoSaxophone	345
5.1.40	instrumenttools.SopranoSaxophone	348
5.1.41	instrumenttools.SopranoVoice	351
5.1.42	instrumenttools.TenorSaxophone	354
5.1.43	instrumenttools.TenorTrombone	357
5.1.44	instrumenttools.TenorVoice	360
5.1.45	instrumenttools.Trumpet	363
5.1.46	instrumenttools.Tuba	366
5.1.47	instrumenttools.UntunedPercussion	369
5.1.48	instrumenttools.Vibraphone	372
5.1.49	instrumenttools.Viola	375
5.1.50	instrumenttools.Violin	378
5.1.51	instrumenttools.WoodwindFingering	381
5.1.52	instrumenttools.Xylophone	385
5.2	Functions	387
5.2.1	instrumenttools.iterate_out_of_range_notes_and_chords	387
5.2.2	instrumenttools.notes_and_chords_are_in_range	388
5.2.3	instrumenttools.notes_and_chords_are_on_expected_clefs	388
5.2.4	instrumenttools.transpose_from_sounding_pitch_to_written_pitch	389
5.2.5	instrumenttools.transpose_from_written_pitch_to_sounding_pitch	389
6	labeltools	391
6.1	Functions	391
6.1.1	labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map	391
6.1.2	labeltools.color_contents_of_container	391
6.1.3	labeltools.color_leaf	392
6.1.4	labeltools.color_leaves_in_expr	392
6.1.5	labeltools.color_measure	393
6.1.6	labeltools.color_measures_with_non_power_of_two_denominators_in_expr	393
6.1.7	labeltools.color_note_head_by_numbered_pitch_class_color_map	394
6.1.8	labeltools.label_leaves_in_expr_with_leaf_depth	394
6.1.9	labeltools.label_leaves_in_expr_with_leaf_duration	395
6.1.10	labeltools.label_leaves_in_expr_with_leaf_durations	395
6.1.11	labeltools.label_leaves_in_expr_with_leaf_indices	396
6.1.12	labeltools.label_leaves_in_expr_with_leaf_numbers	396
6.1.13	labeltools.label_leaves_in_expr_with_named_interval_classes	397
6.1.14	labeltools.label_leaves_in_expr_with_named_intervals	397
6.1.15	labeltools.label_leaves_in_expr_with_numbered_interval_classes	397
6.1.16	labeltools.label_leaves_in_expr_with_numbered_intervals	398
6.1.17	labeltools.label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes	398
6.1.18	labeltools.label_leaves_in_expr_with_pitch_class_numbers	398
6.1.19	labeltools.label_leaves_in_expr_with_pitch_numbers	399
6.1.20	labeltools.label_leaves_in_expr_with_tuplet_depth	399
6.1.21	labeltools.label_leaves_in_expr_with_written_leaf_duration	400
6.1.22	labeltools.label_logical_ties_in_expr_with_logical_tie_duration	400
6.1.23	labeltools.label_logical_ties_in_expr_with_logical_tie_durations	401
6.1.24	labeltools.label_logical_ties_in_expr_with_written_logical_tie_duration	401
6.1.25	labeltools.label_notes_in_expr_with_note_indices	401
6.1.26	labeltools.label_vertical_moments_in_expr_with_interval_class_vectors	402
6.1.27	labeltools.label_vertical_moments_in_expr_with_named_intervals	402
6.1.28	labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes	403
6.1.29	labeltools.label_vertical_moments_in_expr_with_numbered_intervals	403
6.1.30	labeltools.label_vertical_moments_in_expr_with_numbered_pitch_classes	404
6.1.31	labeltools.label_vertical_moments_in_expr_with_pitch_numbers	404
6.1.32	labeltools.remove_markup_from_leaves_in_expr	405

7	layouttools	407
7.1	Concrete classes	407
7.1.1	layouttools.SpacingIndication	407
7.2	Functions	409
7.2.1	layouttools.make_spacing_vector	409
7.2.2	layouttools.set_line_breaks_by_line_duration	409
7.2.3	layouttools.set_line_breaks_by_line_duration_ge	409
7.2.4	layouttools.set_line_breaks_by_line_duration_in_seconds_ge	410
8	lilypondfiletools	413
8.1	Concrete classes	413
8.1.1	lilypondfiletools.Block	413
8.1.2	lilypondfiletools.ContextBlock	415
8.1.3	lilypondfiletools.DateTimeToken	418
8.1.4	lilypondfiletools.LilyPondDimension	420
8.1.5	lilypondfiletools.LilyPondFile	421
8.1.6	lilypondfiletools.LilyPondLanguageToken	425
8.1.7	lilypondfiletools.LilyPondVersionToken	426
8.2	Functions	427
8.2.1	lilypondfiletools.make_basic_lilypond_file	427
8.2.2	lilypondfiletools.make_floating_time_signature_lilypond_file	428
9	markuptools	431
9.1	Concrete classes	431
9.1.1	markuptools.Markup	431
9.1.2	markuptools.MarkupCommand	434
9.1.3	markuptools.MarkupInventory	437
9.1.4	markuptools.MusicGlyph	442
9.2	Functions	444
9.2.1	markuptools.combine_markup_commands	444
9.2.2	markuptools.make_big_centered_page_number_markup	444
9.2.3	markuptools.make_blank_line_markup	445
9.2.4	markuptools.make_centered_title_markup	445
9.2.5	markuptools.make_vertically_adjusted_composer_markup	445
10	mathtools	447
10.1	Concrete classes	447
10.1.1	mathtools.BoundedObject	447
10.1.2	mathtools.Infinity	449
10.1.3	mathtools.NegativeInfinity	451
10.1.4	mathtools.NonreducedFraction	453
10.1.5	mathtools.NonreducedRatio	460
10.1.6	mathtools.Ratio	463
10.2	Functions	465
10.2.1	mathtools.all_are_equal	465
10.2.2	mathtools.all_are_integer_equivalent_exprs	465
10.2.3	mathtools.all_are_integer_equivalent_numbers	465
10.2.4	mathtools.all_are_nonnegative_integer_equivalent_numbers	466
10.2.5	mathtools.all_are_nonnegative_integer_powers_of_two	466
10.2.6	mathtools.all_are_nonnegative_integers	466
10.2.7	mathtools.all_are_numbers	466
10.2.8	mathtools.all_are_pairs	467
10.2.9	mathtools.all_are_pairs_of_types	467
10.2.10	mathtools.all_are_positive_integer_equivalent_numbers	467
10.2.11	mathtools.all_are_positive_integer_powers_of_two	467
10.2.12	mathtools.all_are_positive_integers	468
10.2.13	mathtools.all_are_unequal	468
10.2.14	mathtools.are_relatively_prime	468
10.2.15	mathtools.arithmetic_mean	469

10.2.16	mathtools.binomial_coefficient	469
10.2.17	mathtools.cumulative_products	469
10.2.18	mathtools.cumulative_signed_weights	470
10.2.19	mathtools.cumulative_sums	470
10.2.20	mathtools.cumulative_sums_pairwise	470
10.2.21	mathtools.difference_series	470
10.2.22	mathtools.divide_number_by_ratio	470
10.2.23	mathtools.divisors	471
10.2.24	mathtools.factors	471
10.2.25	mathtools.fraction_to_proper_fraction	472
10.2.26	mathtools.get_shared_numeric_sign	472
10.2.27	mathtools.greatest_common_divisor	472
10.2.28	mathtools.greatest_multiple_less_equal	472
10.2.29	mathtools.greatest_power_of_two_less_equal	473
10.2.30	mathtools.integer_equivalent_number_to_integer	474
10.2.31	mathtools.integer_to_base_k_tuple	474
10.2.32	mathtools.integer_to_binary_string	474
10.2.33	mathtools.is_assignable_integer	474
10.2.34	mathtools.is_dotted_integer	475
10.2.35	mathtools.is_fraction_equivalent_pair	475
10.2.36	mathtools.is_integer_equivalent_expr	476
10.2.37	mathtools.is_integer_equivalent_n_tuple	476
10.2.38	mathtools.is_integer_equivalent_number	476
10.2.39	mathtools.is_integer_equivalent_pair	476
10.2.40	mathtools.is_integer_equivalent_singleton	477
10.2.41	mathtools.is_integer_n_tuple	477
10.2.42	mathtools.is_integer_pair	477
10.2.43	mathtools.is_integer_singleton	477
10.2.44	mathtools.is_n_tuple	478
10.2.45	mathtools.is_negative_integer	478
10.2.46	mathtools.is_nonnegative_integer	478
10.2.47	mathtools.is_nonnegative_integer_equivalent_number	478
10.2.48	mathtools.is_nonnegative_integer_power_of_two	479
10.2.49	mathtools.is_null_tuple	479
10.2.50	mathtools.is_pair	479
10.2.51	mathtools.is_positive_integer	479
10.2.52	mathtools.is_positive_integer_equivalent_number	480
10.2.53	mathtools.is_positive_integer_power_of_two	480
10.2.54	mathtools.is_singleton	480
10.2.55	mathtools.least_common_multiple	481
10.2.56	mathtools.least_multiple_greater_equal	481
10.2.57	mathtools.least_power_of_two_greater_equal	481
10.2.58	mathtools.next_integer_partition	482
10.2.59	mathtools.partition_integer_by_ratio	482
10.2.60	mathtools.partition_integer_into_canonic_parts	483
10.2.61	mathtools.partition_integer_into_halves	484
10.2.62	mathtools.partition_integer_into_parts_less_than_double	484
10.2.63	mathtools.partition_integer_into_units	485
10.2.64	mathtools.remove_powers_of_two	485
10.2.65	mathtools.sign	485
10.2.66	mathtools.weight	486
10.2.67	mathtools.yield_all_compositions_of_integer	486
10.2.68	mathtools.yield_all_partitions_of_integer	486
10.2.69	mathtools.yield_nonreduced_fractions	487
11	metertools	489
11.1	Concrete classes	489
11.1.1	metertools.Meter	489

11.1.2	metertools.MeterManager	497
11.1.3	metertools.MetricAccentKernel	499
12	pitchtools	503
12.1	Abstract classes	503
12.1.1	pitchtools.Interval	503
12.1.2	pitchtools.IntervalClass	505
12.1.3	pitchtools.Pitch	507
12.1.4	pitchtools.PitchClass	511
12.1.5	pitchtools.Segment	514
12.1.6	pitchtools.Set	517
12.1.7	pitchtools.Vector	520
12.2	Concrete classes	523
12.2.1	pitchtools.Accidental	523
12.2.2	pitchtools.IntervalClassSegment	526
12.2.3	pitchtools.IntervalClassSet	529
12.2.4	pitchtools.IntervalClassVector	533
12.2.5	pitchtools.IntervalSegment	537
12.2.6	pitchtools.IntervalSet	540
12.2.7	pitchtools.IntervalVector	544
12.2.8	pitchtools.NamedInterval	548
12.2.9	pitchtools.NamedIntervalClass	553
12.2.10	pitchtools.NamedInversionEquivalentIntervalClass	555
12.2.11	pitchtools.NamedPitch	558
12.2.12	pitchtools.NamedPitchClass	565
12.2.13	pitchtools.NumberedInterval	570
12.2.14	pitchtools.NumberedIntervalClass	573
12.2.15	pitchtools.NumberedInversionEquivalentIntervalClass	575
12.2.16	pitchtools.NumberedPitch	578
12.2.17	pitchtools.NumberedPitchClass	584
12.2.18	pitchtools.NumberedPitchClassColorMap	589
12.2.19	pitchtools.Octave	591
12.2.20	pitchtools.OctaveTranspositionMapping	595
12.2.21	pitchtools.OctaveTranspositionMappingComponent	600
12.2.22	pitchtools.OctaveTranspositionMappingInventory	602
12.2.23	pitchtools.PitchArray	607
12.2.24	pitchtools.PitchArrayCell	613
12.2.25	pitchtools.PitchArrayColumn	616
12.2.26	pitchtools.PitchArrayInventory	619
12.2.27	pitchtools.PitchArrayRow	624
12.2.28	pitchtools.PitchClassSegment	629
12.2.29	pitchtools.PitchClassSet	634
12.2.30	pitchtools.PitchClassTree	639
12.2.31	pitchtools.PitchClassVector	653
12.2.32	pitchtools.PitchRange	656
12.2.33	pitchtools.PitchRangeInventory	660
12.2.34	pitchtools.PitchSegment	665
12.2.35	pitchtools.PitchSet	671
12.2.36	pitchtools.PitchVector	675
12.2.37	pitchtools.TwelveToneRow	678
12.3	Functions	683
12.3.1	pitchtools.apply_accidental_to_named_pitch	683
12.3.2	pitchtools.clef_and_staff_position_number_to_named_pitch	683
12.3.3	pitchtools.contains_subsegment	684
12.3.4	pitchtools.get_named_pitch_from_pitch_carrier	684
12.3.5	pitchtools.get_numbered_pitch_class_from_pitch_carrier	684
12.3.6	pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list	685
12.3.7	pitchtools.instantiate_pitch_and_interval_test_collection	685

12.3.8	pitchtools.inventory_aggregate_subsets	686
12.3.9	pitchtools.iterate_named_pitch_pairs_in_expr	686
12.3.10	pitchtools.list_named_pitches_in_expr	687
12.3.11	pitchtools.list_numbered_interval_numbers_pairwise	687
12.3.12	pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise	688
12.3.13	pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range	689
12.3.14	pitchtools.list_ordered_named_pitch_pairs_from_expr_1_to_expr_2	689
12.3.15	pitchtools.list_pitch_numbers_in_expr	690
12.3.16	pitchtools.list_unordered_named_pitch_pairs_in_expr	690
12.3.17	pitchtools.make_n_middle_c_centered_pitches	690
12.3.18	pitchtools.named_pitch_and_clef_to_staff_position_number	691
12.3.19	pitchtools.numbered_inversion_equivalent_interval_class_dictionary	691
12.3.20	pitchtools.permute_named_pitch_carrier_list_by_twelve_tone_row	691
12.3.21	pitchtools.register_pitch_class_numbers_by_pitch_number_aggregate	691
12.3.22	pitchtools.set_written_pitch_of_pitched_components_in_expr	692
12.3.23	pitchtools.sort_named_pitch_carriers_in_expr	692
12.3.24	pitchtools.spell_numbered_interval_number	692
12.3.25	pitchtools.spell_pitch_number	692
12.3.26	pitchtools.suggest_clef_for_named_pitches	693
12.3.27	pitchtools.transpose_named_pitch_by_numbered_interval_and_respell	693
12.3.28	pitchtools.transpose_pitch_carrier_by_interval	693
12.3.29	pitchtools.transpose_pitch_class_number_to_pitch_number_neighbor	693
12.3.30	pitchtools.transpose_pitch_expr_into_pitch_range	694
12.3.31	pitchtools.transpose_pitch_number_by_octave_transposition_mapping	694
13	quantizationtools	697
13.1	Abstract classes	697
13.1.1	quantizationtools.AttackPointOptimizer	697
13.1.2	quantizationtools.GraceHandler	699
13.1.3	quantizationtools.Heuristic	701
13.1.4	quantizationtools.JobHandler	702
13.1.5	quantizationtools.QEvent	704
13.1.6	quantizationtools.QSchema	706
13.1.7	quantizationtools.QSchemaItem	708
13.1.8	quantizationtools.QTarget	710
13.1.9	quantizationtools.SearchTree	712
13.2	Concrete classes	714
13.2.1	quantizationtools.BeatwiseQSchema	714
13.2.2	quantizationtools.BeatwiseQSchemaItem	719
13.2.3	quantizationtools.BeatwiseQTarget	721
13.2.4	quantizationtools.CollapsingGraceHandler	723
13.2.5	quantizationtools.ConcatenatingGraceHandler	724
13.2.6	quantizationtools.DiscardingGraceHandler	726
13.2.7	quantizationtools.DistanceHeuristic	727
13.2.8	quantizationtools.MeasurewiseAttackPointOptimizer	729
13.2.9	quantizationtools.MeasurewiseQSchema	731
13.2.10	quantizationtools.MeasurewiseQSchemaItem	735
13.2.11	quantizationtools.MeasurewiseQTarget	737
13.2.12	quantizationtools.NaiveAttackPointOptimizer	739
13.2.13	quantizationtools.NullAttackPointOptimizer	740
13.2.14	quantizationtools.ParallelJobHandler	742
13.2.15	quantizationtools.ParallelJobHandlerWorker	743
13.2.16	quantizationtools.PitchedQEvent	745
13.2.17	quantizationtools.QEventProxy	747
13.2.18	quantizationtools.QEventSequence	749
13.2.19	quantizationtools.QGrid	755
13.2.20	quantizationtools.QGridContainer	758
13.2.21	quantizationtools.QGridLeaf	771

13.2.22	quantizationtools.QTargetBeat	777
13.2.23	quantizationtools.QTargetMeasure	780
13.2.24	quantizationtools.QuantizationJob	783
13.2.25	quantizationtools.Quantizer	786
13.2.26	quantizationtools.SerialJobHandler	790
13.2.27	quantizationtools.SilentQEvent	791
13.2.28	quantizationtools.TerminalQEvent	793
13.2.29	quantizationtools.UnweightedSearchTree	795
13.2.30	quantizationtools.WeightedSearchTree	798
13.3	Functions	800
13.3.1	quantizationtools.make_test_time_segments	800
14	rhythmmakertools	801
14.1	Abstract classes	801
14.1.1	rhythmmakertools.RhythmMaker	801
14.2	Concrete classes	803
14.2.1	rhythmmakertools.BeamSpecifier	803
14.2.2	rhythmmakertools.BurnishSpecifier	806
14.2.3	rhythmmakertools.DurationSpellingSpecifier	811
14.2.4	rhythmmakertools.EvenRunRhythmMaker	814
14.2.5	rhythmmakertools.ExampleWrapper	818
14.2.6	rhythmmakertools.GalleryMaker	819
14.2.7	rhythmmakertools.InciseSpecifier	821
14.2.8	rhythmmakertools.IncisedRhythmMaker	824
14.2.9	rhythmmakertools.NoteRhythmMaker	828
14.2.10	rhythmmakertools.RestRhythmMaker	831
14.2.11	rhythmmakertools.SkipRhythmMaker	834
14.2.12	rhythmmakertools.Talea	837
14.2.13	rhythmmakertools.TaleaRhythmMaker	839
14.2.14	rhythmmakertools.TieSpecifier	849
14.2.15	rhythmmakertools.TupletRhythmMaker	851
14.2.16	rhythmmakertools.TupletSpellingSpecifier	855
14.3	Functions	856
14.3.1	rhythmmakertools.make_lilypond_file	856
15	rhythmtreetools	857
15.1	Abstract classes	857
15.1.1	rhythmtreetools.RhythmTreeNode	857
15.2	Concrete classes	863
15.2.1	rhythmtreetools.RhythmTreeContainer	863
15.2.2	rhythmtreetools.RhythmTreeLeaf	877
15.2.3	rhythmtreetools.RhythmTreeParser	883
15.3	Functions	886
15.3.1	rhythmtreetools.parse_rtm_syntax	886
16	schemetools	887
16.1	Concrete classes	887
16.1.1	schemetools.Scheme	887
16.1.2	schemetools.SchemeAssociativeList	890
16.1.3	schemetools.SchemeColor	892
16.1.4	schemetools.SchemeMoment	894
16.1.5	schemetools.SchemePair	897
16.1.6	schemetools.SchemeVector	899
16.1.7	schemetools.SchemeVectorConstant	901
17	scoretools	905
17.1	Abstract classes	905
17.1.1	scoretools.Component	905
17.1.2	scoretools.Leaf	907

17.2	Concrete classes	909
17.2.1	scoretools.Chord	909
17.2.2	scoretools.Cluster	913
17.2.3	scoretools.Container	919
17.2.4	scoretools.Context	925
17.2.5	scoretools.FixedDurationContainer	932
17.2.6	scoretools.FixedDurationTuplet	938
17.2.7	scoretools.GraceContainer	955
17.2.8	scoretools.Measure	962
17.2.9	scoretools.MultimeasureRest	972
17.2.10	scoretools.Note	974
17.2.11	scoretools.NoteHead	977
17.2.12	scoretools.NoteHeadInventory	980
17.2.13	scoretools.Rest	985
17.2.14	scoretools.Score	987
17.2.15	scoretools.Skip	995
17.2.16	scoretools.Staff	997
17.2.17	scoretools.StaffGroup	1004
17.2.18	scoretools.Tuplet	1011
17.2.19	scoretools.Voice	1028
17.3	Functions	1034
17.3.1	scoretools.append_spacer_skip_to_underfull_measure	1034
17.3.2	scoretools.append_spacer_skips_to_underfull_measures_in_expr	1035
17.3.3	scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr	1035
17.3.4	scoretools.extend_measures_in_expr_and_apply_full_measure_tuplets	1035
17.3.5	scoretools.fill_measures_in_expr_with_full_measure_spacer_skips	1036
17.3.6	scoretools.fill_measures_in_expr_with_minimal_number_of_notes	1036
17.3.7	scoretools.fill_measures_in_expr_with_repeated_notes	1036
17.3.8	scoretools.fill_measures_in_expr_with_time_signature_denominator_notes	1036
17.3.9	scoretools.get_measure_that_starts_with_container	1036
17.3.10	scoretools.get_measure_that_stops_with_container	1037
17.3.11	scoretools.get_next_measure_from_component	1037
17.3.12	scoretools.get_one_indexed_measure_number_in_expr	1037
17.3.13	scoretools.get_previous_measure_from_component	1038
17.3.14	scoretools.make_empty_piano_score	1038
17.3.15	scoretools.make_leaves	1038
17.3.16	scoretools.make_leaves_from_talea	1041
17.3.17	scoretools.make_multimeasure_rests	1042
17.3.18	scoretools.make_multiplied_quarter_notes	1042
17.3.19	scoretools.make_notes	1042
17.3.20	scoretools.make_notes_with_multiplied_durations	1043
17.3.21	scoretools.make_percussion_note	1043
17.3.22	scoretools.make_piano_score_from_leaves	1044
17.3.23	scoretools.make_piano_sketch_score_from_leaves	1044
17.3.24	scoretools.make_repeated_notes	1044
17.3.25	scoretools.make_repeated_notes_from_time_signature	1045
17.3.26	scoretools.make_repeated_notes_from_time_signatures	1045
17.3.27	scoretools.make_repeated_notes_with_shorter_notes_at_end	1045
17.3.28	scoretools.make_repeated_rests_from_time_signatures	1046
17.3.29	scoretools.make_repeated_skips_from_time_signatures	1046
17.3.30	scoretools.make_rests	1046
17.3.31	scoretools.make_rhythmic_sketch_staff	1047
17.3.32	scoretools.make_skips_with_multiplied_durations	1047
17.3.33	scoretools.make_spacer_skip_measures	1047
17.3.34	scoretools.make_tied_leaf	1047
17.3.35	scoretools.move_full_measure_tuplet_prolation_to_measure_time_signature	1048
17.3.36	scoretools.move_measure_prolation_to_full_measure_tuplet	1049
17.3.37	scoretools.scale_measure_denominator_and_adjust_measure_contents	1049

17.3.38	scoretools.set_measure_denominator_and_adjust_numerator	1049
18	selectiontools	1051
18.1	Concrete classes	1051
18.1.1	selectiontools.ContiguousSelection	1051
18.1.2	selectiontools.Descendants	1054
18.1.3	selectiontools.Lineage	1056
18.1.4	selectiontools.LogicalTie	1059
18.1.5	selectiontools.Parentage	1063
18.1.6	selectiontools.Selection	1067
18.1.7	selectiontools.SelectionInventory	1069
18.1.8	selectiontools.SimultaneousSelection	1074
18.1.9	selectiontools.SliceSelection	1076
18.1.10	selectiontools.VerticalMoment	1079
19	sequencetools	1083
19.1	Concrete classes	1083
19.1.1	sequencetools.Sequence	1083
19.2	Functions	1088
19.2.1	sequencetools.flatten_sequence	1088
19.2.2	sequencetools.increase_elements	1089
19.2.3	sequencetools.interlace_sequences	1089
19.2.4	sequencetools.iterate_sequence_boustrophedon	1089
19.2.5	sequencetools.iterate_sequence_nwise	1090
19.2.6	sequencetools.join_subsequences	1091
19.2.7	sequencetools.join_subsequences_by_sign_of_elements	1091
19.2.8	sequencetools.negate_elements	1092
19.2.9	sequencetools.overwrite_elements	1093
19.2.10	sequencetools.partition_sequence_by_counts	1093
19.2.11	sequencetools.partition_sequence_by_ratio_of_lengths	1094
19.2.12	sequencetools.partition_sequence_by_ratio_of_weights	1094
19.2.13	sequencetools.partition_sequence_by_restricted_growth_function	1095
19.2.14	sequencetools.partition_sequence_by_sign_of_elements	1095
19.2.15	sequencetools.partition_sequence_by_value_of_elements	1096
19.2.16	sequencetools.partition_sequence_by_weights	1096
19.2.17	sequencetools.permute_sequence	1098
19.2.18	sequencetools.remove_elements	1098
19.2.19	sequencetools.remove_repeated_elements	1099
19.2.20	sequencetools.remove_subsequence_of_weight_at_index	1100
19.2.21	sequencetools.repeat_elements	1100
19.2.22	sequencetools.repeat_sequence	1101
19.2.23	sequencetools.repeat_sequence_to_length	1101
19.2.24	sequencetools.repeat_sequence_to_weight	1101
19.2.25	sequencetools.replace_elements	1102
19.2.26	sequencetools.retain_elements	1103
19.2.27	sequencetools.reverse_sequence	1104
19.2.28	sequencetools.rotate_sequence	1104
19.2.29	sequencetools.splice_between_elements	1104
19.2.30	sequencetools.split_sequence	1105
19.2.31	sequencetools.sum_consecutive_elements_by_sign	1105
19.2.32	sequencetools.sum_elements	1106
19.2.33	sequencetools.truncate_sequence	1106
19.2.34	sequencetools.yield_all_combinations_of_elements	1107
19.2.35	sequencetools.yield_all_k_ary_sequences_of_length	1108
19.2.36	sequencetools.yield_all_pairs_between_sequences	1108
19.2.37	sequencetools.yield_all_partitions_of_sequence	1108
19.2.38	sequencetools.yield_all_permutations_of_sequence	1109
19.2.39	sequencetools.yield_all_permutations_of_sequence_in_orbit	1109

19.2.40	sequencetools.yield_all_restricted_growth_functions_of_length	1109
19.2.41	sequencetools.yield_all_rotations_of_sequence	1110
19.2.42	sequencetools.yield_all_set_partitions_of_sequence	1110
19.2.43	sequencetools.yield_all_subsequences_of_sequence	1110
19.2.44	sequencetools.yield_all_unordered_pairs_of_sequence	1111
19.2.45	sequencetools.yield_outer_product_of_sequences	1111
19.2.46	sequencetools.zip_sequences	1112
20	sievetools	1113
20.1	Concrete classes	1113
20.1.1	sievetools.BaseResidueClass	1113
20.1.2	sievetools.ResidueClass	1115
20.1.3	sievetools.Sieve	1118
21	spannertools	1121
21.1	Concrete classes	1121
21.1.1	spannertools.Beam	1121
21.1.2	spannertools.ComplexBeam	1124
21.1.3	spannertools.ComplexTrillSpanner	1127
21.1.4	spannertools.Crescendo	1129
21.1.5	spannertools.Decrescendo	1133
21.1.6	spannertools.DuratedComplexBeam	1137
21.1.7	spannertools.GeneralizedBeam	1142
21.1.8	spannertools.Glissando	1145
21.1.9	spannertools.Hairpin	1147
21.1.10	spannertools.HiddenStaffSpanner	1151
21.1.11	spannertools.HorizontalBracketSpanner	1153
21.1.12	spannertools.MeasuredComplexBeam	1155
21.1.13	spannertools.MultipartBeam	1159
21.1.14	spannertools.OctavationSpanner	1161
21.1.15	spannertools.PhrasingSlur	1164
21.1.16	spannertools.PianoPedalSpanner	1167
21.1.17	spannertools.Slur	1170
21.1.18	spannertools.Spanner	1173
21.1.19	spannertools.StaffLinesSpanner	1175
21.1.20	spannertools.TextSpanner	1177
21.1.21	spannertools.Tie	1179
21.1.22	spannertools.TrillSpanner	1182
21.2	Functions	1184
21.2.1	spannertools.make_colored_text_spanner_with_nibs	1184
21.2.2	spannertools.make_dynamic_spanner_below_with_nib_at_right	1184
21.2.3	spannertools.make_solid_text_spanner_with_nib	1185
22	stringtools	1187
22.1	Functions	1187
22.1.1	stringtools.add_terminal_newlines	1187
22.1.2	stringtools.arg_to_bidirectional_direction_string	1187
22.1.3	stringtools.arg_to_bidirectional_lilypond_symbol	1187
22.1.4	stringtools.arg_to_tridirectional_direction_string	1188
22.1.5	stringtools.arg_to_tridirectional_lilypond_symbol	1188
22.1.6	stringtools.arg_to_tridirectional_ordinal_constant	1189
22.1.7	stringtools.capitalize_start	1189
22.1.8	stringtools.delimit_words	1189
22.1.9	stringtools.format_input_lines_as_doc_string	1190
22.1.10	stringtools.format_input_lines_as_regression_test	1190
22.1.11	stringtools.is_dash_case	1191
22.1.12	stringtools.is_dash_case_file_name	1191
22.1.13	stringtools.is_lower_camel_case	1191
22.1.14	stringtools.is_snake_case	1192

22.1.15	stringtools.is_snake_case_file_name	1192
22.1.16	stringtools.is_snake_case_file_name_with_extension	1192
22.1.17	stringtools.is_snake_case_package_name	1192
22.1.18	stringtools.is_space_delimited_lowercase	1193
22.1.19	stringtools.is_string	1193
22.1.20	stringtools.is_upper_camel_case	1193
22.1.21	stringtools.pluralize	1193
22.1.22	stringtools.snake_case_to_lower_camel_case	1194
22.1.23	stringtools.snake_case_to_upper_camel_case	1194
22.1.24	stringtools.space_delimited_lowercase_to_upper_camel_case	1194
22.1.25	stringtools.strip_diacritics	1194
22.1.26	stringtools.to_accent_free_snake_case	1194
22.1.27	stringtools.to_dash_case	1195
22.1.28	stringtools.to_snake_case	1195
22.1.29	stringtools.to_space_delimited_lowercase	1196
22.1.30	stringtools.to_upper_camel_case	1196
22.1.31	stringtools.upper_camel_case_to_snake_case	1196
22.1.32	stringtools.upper_camel_case_to_space_delimited_lowercase	1197
23	templatetools	1199
23.1	Concrete classes	1199
23.1.1	templatetools.GroupedRhythmicStavesScoreTemplate	1199
23.1.2	templatetools.GroupedStavesScoreTemplate	1201
23.1.3	templatetools.StringOrchestraScoreTemplate	1203
23.1.4	templatetools.StringQuartetScoreTemplate	1205
23.1.5	templatetools.TwoStaffPianoScoreTemplate	1206
24	timespantools	1209
24.1	Abstract classes	1209
24.1.1	timespantools.TimeRelation	1209
24.2	Concrete classes	1211
24.2.1	timespantools.AnnotatedTimespan	1211
24.2.2	timespantools.CompoundInequality	1232
24.2.3	timespantools.OffsetTimespanTimeRelation	1237
24.2.4	timespantools.SimpleInequality	1240
24.2.5	timespantools.Timespan	1242
24.2.6	timespantools.TimespanInventory	1262
24.2.7	timespantools.TimespanTimespanTimeRelation	1294
24.3	Functions	1299
24.3.1	timespantools.offset_happens_after_timespan_starts	1299
24.3.2	timespantools.offset_happens_after_timespan_stops	1299
24.3.3	timespantools.offset_happens_before_timespan_starts	1300
24.3.4	timespantools.offset_happens_before_timespan_stops	1301
24.3.5	timespantools.offset_happens_during_timespan	1301
24.3.6	timespantools.offset_happens_when_timespan_starts	1302
24.3.7	timespantools.offset_happens_when_timespan_stops	1302
24.3.8	timespantools.timespan_2_contains_timespan_1_improperly	1302
24.3.9	timespantools.timespan_2_curtails_timespan_1	1303
24.3.10	timespantools.timespan_2_delays_timespan_1	1303
24.3.11	timespantools.timespan_2_happens_during_timespan_1	1304
24.3.12	timespantools.timespan_2_intersects_timespan_1	1304
24.3.13	timespantools.timespan_2_is_congruent_to_timespan_1	1304
24.3.14	timespantools.timespan_2_overlaps_all_of_timespan_1	1305
24.3.15	timespantools.timespan_2_overlaps_only_start_of_timespan_1	1305
24.3.16	timespantools.timespan_2_overlaps_only_stop_of_timespan_1	1305
24.3.17	timespantools.timespan_2_overlaps_start_of_timespan_1	1306
24.3.18	timespantools.timespan_2_overlaps_stop_of_timespan_1	1306
24.3.19	timespantools.timespan_2_starts_after_timespan_1_starts	1306

24.3.20	timespantools.timespan_2_starts_after_timespan_1_stops	1307
24.3.21	timespantools.timespan_2_starts_before_timespan_1_starts	1307
24.3.22	timespantools.timespan_2_starts_before_timespan_1_stops	1307
24.3.23	timespantools.timespan_2_starts_during_timespan_1	1308
24.3.24	timespantools.timespan_2_starts_when_timespan_1_starts	1309
24.3.25	timespantools.timespan_2_starts_when_timespan_1_stops	1309
24.3.26	timespantools.timespan_2_stops_after_timespan_1_starts	1309
24.3.27	timespantools.timespan_2_stops_after_timespan_1_stops	1310
24.3.28	timespantools.timespan_2_stops_before_timespan_1_starts	1310
24.3.29	timespantools.timespan_2_stops_before_timespan_1_stops	1310
24.3.30	timespantools.timespan_2_stops_during_timespan_1	1311
24.3.31	timespantools.timespan_2_stops_when_timespan_1_starts	1311
24.3.32	timespantools.timespan_2_stops_when_timespan_1_stops	1311
24.3.33	timespantools.timespan_2_trisects_timespan_1	1311
25	tonalanalysistools	1313
25.1	Concrete classes	1313
25.1.1	tonalanalysistools.ChordExtent	1313
25.1.2	tonalanalysistools.ChordInversion	1315
25.1.3	tonalanalysistools.ChordOmission	1317
25.1.4	tonalanalysistools.ChordQuality	1318
25.1.5	tonalanalysistools.ChordSuspension	1319
25.1.6	tonalanalysistools.Mode	1321
25.1.7	tonalanalysistools.RomanNumeral	1323
25.1.8	tonalanalysistools.RootedChordClass	1325
25.1.9	tonalanalysistools.RootlessChordClass	1331
25.1.10	tonalanalysistools.Scale	1335
25.1.11	tonalanalysistools.ScaleDegree	1341
25.1.12	tonalanalysistools.TonalAnalysisAgent	1344
25.2	Functions	1347
25.2.1	tonalanalysistools.select	1347
26	topleveltools	1349
26.1	Functions	1349
26.1.1	topleveltools.attach	1349
26.1.2	topleveltools.detach	1349
26.1.3	topleveltools.graph	1349
26.1.4	topleveltools.inspect	1350
26.1.5	topleveltools.iterate	1350
26.1.6	topleveltools.mutate	1350
26.1.7	topleveltools.new	1351
26.1.8	topleveltools.override	1351
26.1.9	topleveltools.parse	1351
26.1.10	topleveltools.persist	1351
26.1.11	topleveltools.play	1351
26.1.12	topleveltools.select	1352
26.1.13	topleveltools.set	1352
26.1.14	topleveltools.show	1352
II	Demos and example packages	1353
27	desordre	1355
27.1	Functions	1355
27.1.1	desordre.make_desordre_cell	1355
27.1.2	desordre.make_desordre_lilypond_file	1355
27.1.3	desordre.make_desordre_measure	1355
27.1.4	desordre.make_desordre_pitches	1355
27.1.5	desordre.make_desordre_score	1355

27.1.6	desordre.make_desordre_staff	1355
28	ferneyhough	1357
28.1	Functions	1357
28.1.1	ferneyhough.configure_lilypond_file	1357
28.1.2	ferneyhough.configure_score	1357
28.1.3	ferneyhough.make_lilypond_file	1357
28.1.4	ferneyhough.make_nested_tuplet	1357
28.1.5	ferneyhough.make_row_of_nested_tuplets	1357
28.1.6	ferneyhough.make_rows_of_nested_tuplets	1357
28.1.7	ferneyhough.make_score	1357
29	mozart	1359
29.1	Functions	1359
29.1.1	mozart.choose_mozart_measures	1359
29.1.2	mozart.make_mozart_lilypond_file	1359
29.1.3	mozart.make_mozart_measure	1359
29.1.4	mozart.make_mozart_measure_corpus	1359
29.1.5	mozart.make_mozart_score	1359
30	part	1361
30.1	Concrete classes	1361
30.1.1	part.PartCantusScoreTemplate	1361
30.2	Functions	1362
30.2.1	part.add_bell_music_to_score	1362
30.2.2	part.add_string_music_to_score	1362
30.2.3	part.apply_bowing_marks	1362
30.2.4	part.apply_dynamics	1362
30.2.5	part.apply_expressive_marks	1362
30.2.6	part.apply_final_bar_lines	1363
30.2.7	part.apply_page_breaks	1363
30.2.8	part.apply_rehearsal_marks	1363
30.2.9	part.configure_lilypond_file	1363
30.2.10	part.configure_score	1363
30.2.11	part.create_pitch_contour_reservoir	1363
30.2.12	part.durate_pitch_contour_reservoir	1363
30.2.13	part.edit_bass_voice	1363
30.2.14	part.edit_cello_voice	1363
30.2.15	part.edit_first_violin_voice	1363
30.2.16	part.edit_second_violin_voice	1364
30.2.17	part.edit_viola_voice	1364
30.2.18	part.make_part_lilypond_file	1364
30.2.19	part.shadow_pitch_contour_reservoir	1364
III	Abjad internal packages	1365
31	abctools	1367
31.1	Abstract classes	1367
31.1.1	abctools.ContextManager	1367
31.1.2	abctools.Parser	1369
31.2	Concrete classes	1371
31.2.1	abctools.AbjadObject	1371
31.2.2	abctools.AbjadValueObject	1372
32	abjadbooktools	1375
32.1	Abstract classes	1375
32.1.1	abjadbooktools.OutputFormat	1375
32.2	Concrete classes	1377

32.2.1	abjadbooktools.AbjadBookProcessor	1377
32.2.2	abjadbooktools.AbjadBookScript	1379
32.2.3	abjadbooktools.CodeBlock	1381
32.2.4	abjadbooktools.HTMLOutputFormat	1383
32.2.5	abjadbooktools.LaTeXOutputFormat	1385
32.2.6	abjadbooktools.ReSTOutputFormat	1387
33	developerscripttools	1389
33.1	Abstract classes	1389
33.1.1	developerscripttools.DeveloperScript	1389
33.1.2	developerscripttools.DirectoryScript	1392
33.2	Concrete classes	1394
33.2.1	developerscripttools.AbjDevScript	1394
33.2.2	developerscripttools.AbjGrepScript	1397
33.2.3	developerscripttools.BuildApiScript	1400
33.2.4	developerscripttools.CleanScript	1403
33.2.5	developerscripttools.CountLinewidthsScript	1406
33.2.6	developerscripttools.CountToolsScript	1409
33.2.7	developerscripttools.MakeNewClassTemplateScript	1412
33.2.8	developerscripttools.MakeNewFunctionTemplateScript	1414
33.2.9	developerscripttools.PyTestScript	1417
33.2.10	developerscripttools.RenameModulesScript	1420
33.2.11	developerscripttools.ReplaceInFilesScript	1423
33.2.12	developerscripttools.RunDoctestsScript	1426
33.2.13	developerscripttools.TestAndRebuildScript	1429
33.3	Functions	1431
33.3.1	developerscripttools.get_developer_script_classes	1431
33.3.2	developerscripttools.run_abjadbook	1431
33.3.3	developerscripttools.run_ajv	1431
34	documentationtools	1433
34.1	Abstract classes	1433
34.1.1	documentationtools.GraphvizObject	1433
34.1.2	documentationtools.ReSTDirective	1435
34.2	Concrete classes	1444
34.2.1	documentationtools.AbjadAPIGenerator	1444
34.2.2	documentationtools.ClassCrawler	1446
34.2.3	documentationtools.ClassDocmenter	1447
34.2.4	documentationtools.Docmenter	1450
34.2.5	documentationtools.FunctionCrawler	1452
34.2.6	documentationtools.FunctionDocmenter	1453
34.2.7	documentationtools.GraphvizEdge	1455
34.2.8	documentationtools.GraphvizField	1457
34.2.9	documentationtools.GraphvizGraph	1461
34.2.10	documentationtools.GraphvizGroup	1473
34.2.11	documentationtools.GraphvizNode	1482
34.2.12	documentationtools.GraphvizSubgraph	1491
34.2.13	documentationtools.InheritanceGraph	1501
34.2.14	documentationtools.ModuleCrawler	1504
34.2.15	documentationtools.Pipe	1506
34.2.16	documentationtools.ReSTAutodocDirective	1508
34.2.17	documentationtools.ReSTAutosummaryDirective	1518
34.2.18	documentationtools.ReSTAutosummaryItem	1527
34.2.19	documentationtools.ReSTDdocument	1531
34.2.20	documentationtools.ReSTHeading	1541
34.2.21	documentationtools.ReSTHorizontalRule	1545
34.2.22	documentationtools.ReSTInheritanceDiagram	1549
34.2.23	documentationtools.ReSTLineageDirective	1558

34.2.24	documentationtools.ReSTOnlyDirective	1568
34.2.25	documentationtools.ReSTParagraph	1578
34.2.26	documentationtools.ReSTTOCDirective	1582
34.2.27	documentationtools.ReSTTOCItem	1591
34.2.28	documentationtools.ToolsPackageDocumenter	1595
34.3	Functions	1597
34.3.1	documentationtools.compare_images	1597
34.3.2	documentationtools.list_all_abjad_classes	1597
34.3.3	documentationtools.list_all_abjad_functions	1598
34.3.4	documentationtools.list_all_classes	1598
34.3.5	documentationtools.list_all_experimental_classes	1598
34.3.6	documentationtools.list_all_scoremanager_classes	1598
34.3.7	documentationtools.list_all_scoremanager_functions	1598
34.3.8	documentationtools.make_ligeti_example_lilypond_file	1598
34.3.9	documentationtools.make_reference_manual_graphviz_graph	1598
34.3.10	documentationtools.make_reference_manual_lilypond_file	1599
34.3.11	documentationtools.make_text_alignment_example_lilypond_file	1599
35	exceptiontools	1601
35.1	Concrete classes	1601
35.1.1	exceptiontools.AssignabilityError	1601
35.1.2	exceptiontools.ExtraSpannerError	1602
35.1.3	exceptiontools.ImpreciseTempoError	1603
35.1.4	exceptiontools.LilyPondParserError	1604
35.1.5	exceptiontools.MissingMeasureError	1605
35.1.6	exceptiontools.MissingSpannerError	1606
35.1.7	exceptiontools.MissingTempoError	1607
35.1.8	exceptiontools.OverfullContainerError	1608
35.1.9	exceptiontools.PartitionError	1609
35.1.10	exceptiontools.SchemeParserFinishedError	1610
35.1.11	exceptiontools.UnboundedTimeIntervalError	1611
35.1.12	exceptiontools.UnderfullContainerError	1612
36	lilypondnametools	1615
36.1	Concrete classes	1615
36.1.1	lilypondnametools.LilyPondContextSetting	1615
36.1.2	lilypondnametools.LilyPondGrobNameManager	1617
36.1.3	lilypondnametools.LilyPondGrobOverride	1618
36.1.4	lilypondnametools.LilyPondNameManager	1621
36.1.5	lilypondnametools.LilyPondSettingNameManager	1622
37	lilypondparsertools	1625
37.1	Abstract classes	1625
37.1.1	lilypondparsertools.Music	1625
37.1.2	lilypondparsertools.SimultaneousMusic	1627
37.2	Concrete classes	1628
37.2.1	lilypondparsertools.ContextSpeccedMusic	1628
37.2.2	lilypondparsertools.GuileProxy	1630
37.2.3	lilypondparsertools.LilyPondDuration	1632
37.2.4	lilypondparsertools.LilyPondEvent	1633
37.2.5	lilypondparsertools.LilyPondFraction	1634
37.2.6	lilypondparsertools.LilyPondGrammarGenerator	1636
37.2.7	lilypondparsertools.LilyPondLexicalDefinition	1637
37.2.8	lilypondparsertools.LilyPondParser	1640
37.2.9	lilypondparsertools.LilyPondSyntacticalDefinition	1650
37.2.10	lilypondparsertools.ReducedLyParser	1668
37.2.11	lilypondparsertools.SchemeParser	1674
37.2.12	lilypondparsertools.SequentialMusic	1678
37.2.13	lilypondparsertools.SyntaxNode	1679

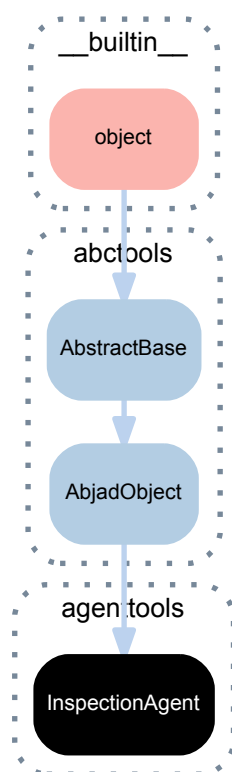
37.3	Functions	1680
37.3.1	lilypondparsertools.parse_reduced_ly_syntax	1680
38	systemtools	1683
38.1	Abstract classes	1683
38.1.1	systemtools.Configuration	1683
38.2	Concrete classes	1685
38.2.1	systemtools.AbjadConfiguration	1685
38.2.2	systemtools.AttributeDetail	1689
38.2.3	systemtools.AttributeManifest	1691
38.2.4	systemtools.BenchmarkScoreMaker	1692
38.2.5	systemtools.FilesystemState	1696
38.2.6	systemtools.ForbidUpdate	1698
38.2.7	systemtools.IOManager	1699
38.2.8	systemtools.ImportManager	1703
38.2.9	systemtools.LilyPondFormatBundle	1704
38.2.10	systemtools.LilyPondFormatManager	1706
38.2.11	systemtools.Memoize	1708
38.2.12	systemtools.NullContextManager	1710
38.2.13	systemtools.ProgressIndicator	1711
38.2.14	systemtools.RedirectedStreams	1713
38.2.15	systemtools.StorageFormatManager	1715
38.2.16	systemtools.StorageFormatSpecification	1717
38.2.17	systemtools.TemporaryDirectoryChange	1719
38.2.18	systemtools.TestManager	1720
38.2.19	systemtools.Timer	1722
38.2.20	systemtools.UpdateManager	1724
38.2.21	systemtools.WellformednessManager	1725
38.3	Functions	1727
38.3.1	systemtools.requires	1727
38.3.2	systemtools.run_abjad	1728
Index		1729

Part I

Core composition packages

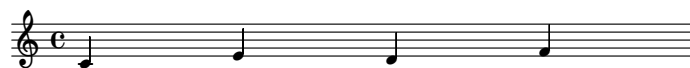
1.1 Concrete classes

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

Bases

- abctools.AbjadObject

- `abctools.AbjadObject.AbstractBase`
- `__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)
```

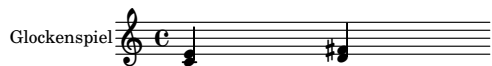


Returns named pitch.

`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*)

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

Returns boolean.

`InspectionAgent.has_indicator` (*prototype=None*)

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

Returns boolean.

`InspectionAgent.is_bar_line_crossing` ()

Is 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*)

Is 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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

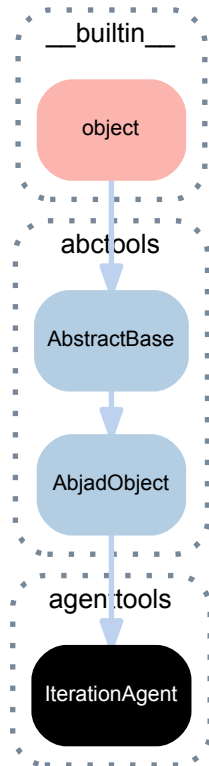
`(AbjadObject).__ne__(expr)`

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

Returns boolean.

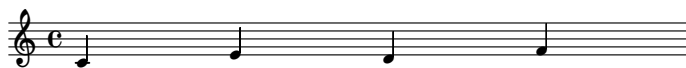
`(AbjadObject).__repr__()`
 Gets interpreter representation of Abjad object.
 Returns string.

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`
- `abctools.AbjadObject.AbstractBase`
- `__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*) *re-*

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("c'8 d'8")
Note("d'8")
```

```
Voice("g'8 a'8")
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("b'8 c'8")
Note("b'8")
Voice("e'8 f'8")
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*:

```
>>> pairs = [(3, 8), (5, 16), (5, 16)]
>>> measures = scoretools.make_spacer_skip_measures(pairs)
>>> 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("c'4. d'4 e'16 f'4 g'16")
```

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



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



```
>>> 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)
```

```
>>> staff_group = StaffGroup([])
>>> staff_group.context_name = 'PianoStaff'
>>> staff_group.append(Staff("a'4 g'4"))
>>> staff_group.append(Staff(r"""\clef "bass" f'8 e'8 d'8 c'8"""))
>>> score.append(staff_group)
```



```
>>> 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(staff_group).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(staff_group).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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__** ()

Hashes Abjad object.

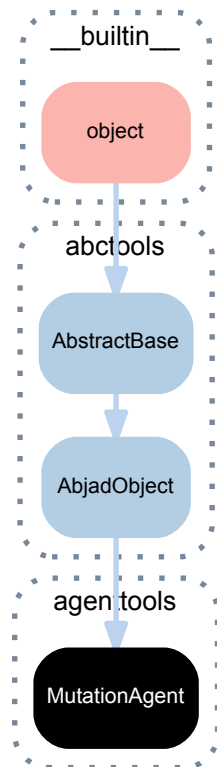
Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

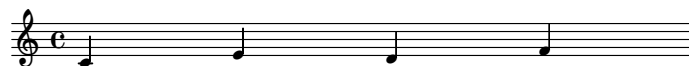
`(AbjadObject).__repr__()`
 Gets interpreter representation of Abjad object.
 Returns string.

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`
- `abctools.AbjadObject.AbstractBase`
- `__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)
```



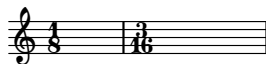
Preserves both hairpin and slur.

Returns none.

`MutationAgent.replace_measure_contents` (*new_contents*)

Replaces contents of measures in client with *new_contents*.

```
>>> pairs = [(1, 8), (3, 16)]
>>> measures = scoretools.make_spacer_skip_measures(pairs)
>>> staff = Staff(measures)
>>> show(staff)
```



```
>>> notes = [Note("c'16"), Note("d'16"), Note("e'16"), Note("f'16")]
>>> mutate(staff).replace_measure_contents(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.

`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.

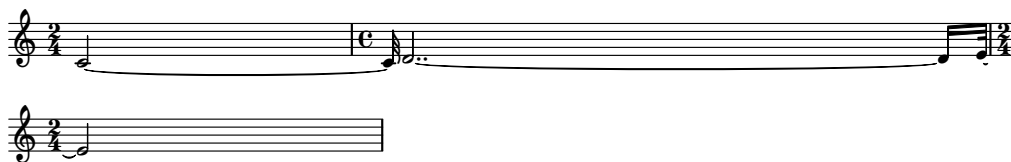
`MutationAgent.rewrite_meter(meter, boundary_depth=None, initial_offset=None, maximum_dot_count=None)`

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)
```

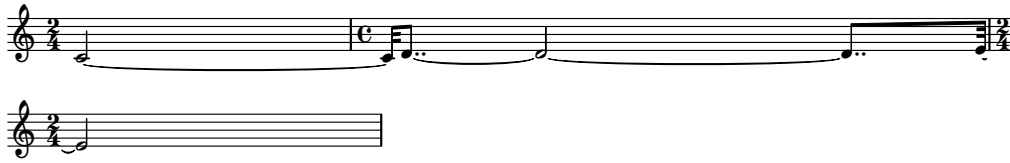
```
>>> show(staff)
```



```
>>> 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)
```

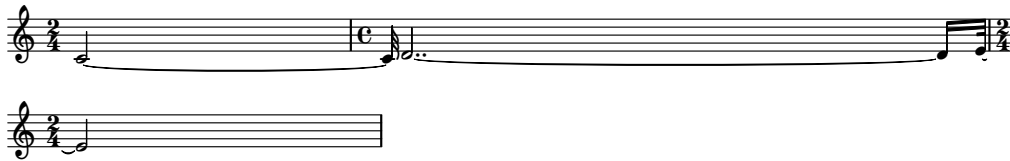
```
>>> show(staff)
```



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

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

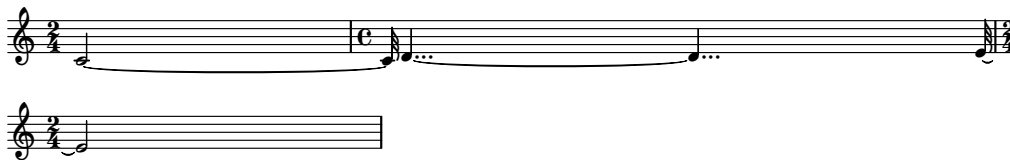
```
>>> show(staff)
```



```
>>> 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)
```



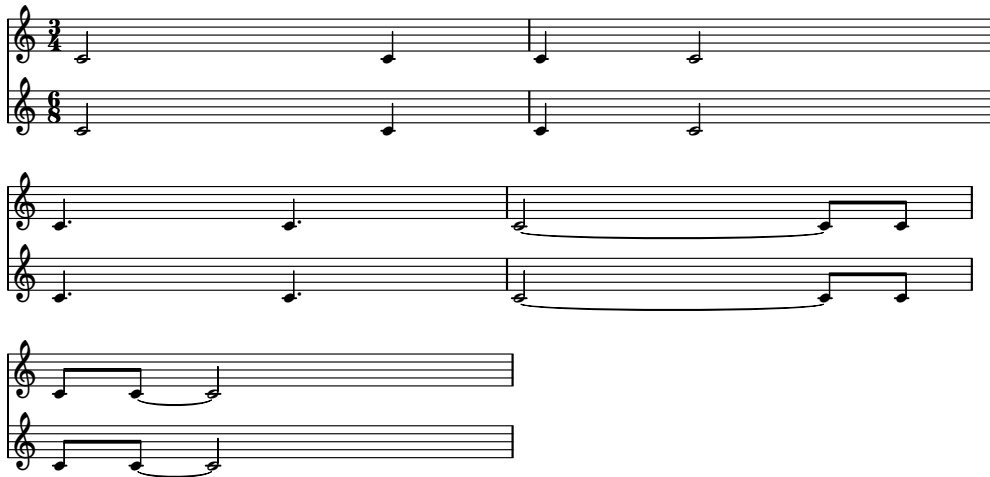
Example 5. Comparison of 3/4 and 6/8, at *boundary_depths* of 0 and 1:

```
>>> 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.remove_commands.extend(engravers)
>>> score[0].consists_commands.extend(engravers)
>>> score[1].consists_commands.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)
```

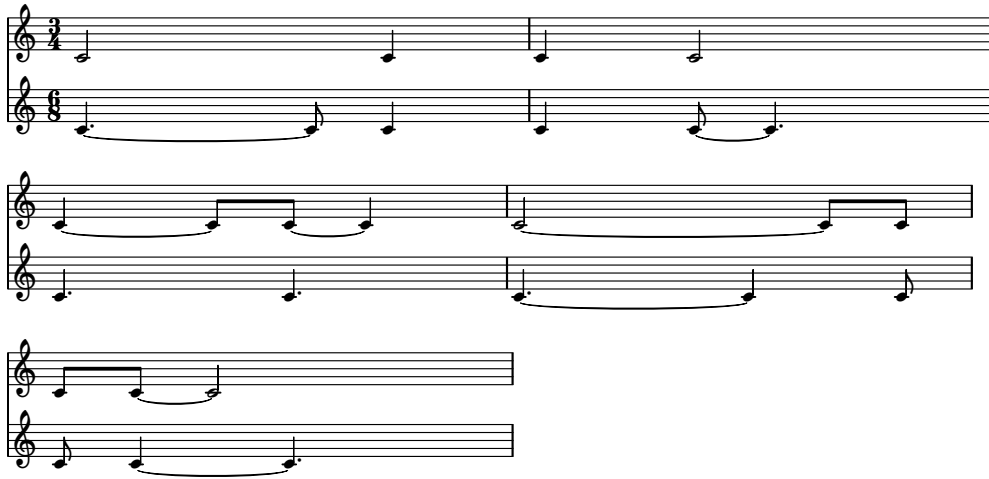
```
>>> show(score)
```



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)
```

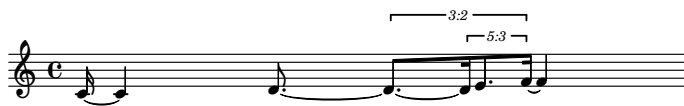


Note that the two time signatures are much more clearly disambiguated above.

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)
```



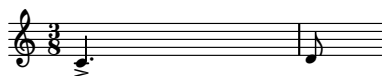
```
>>> mutate(staff[1]).scale(Multiplier(3, 2))  
>>> show(staff)
```

**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)
```



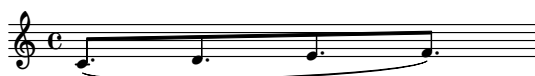
```
>>> logical_tie = inspect_(staff[0]).get_logical_tie()
>>> logical_tie = mutate(logical_tie).scale(Multiplier(3, 2))
>>> show(staff)
```

**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)
```



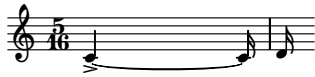
```
>>> mutate(staff[1]).scale(Multiplier(5, 4))
>>> show(staff)
```

**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)
```

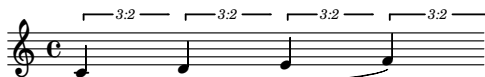


Example 3c. Scale container by tuplet-generating multiplier:

```
>>> 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)
```

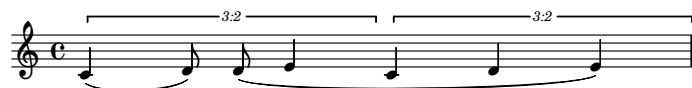


Example 5. Split tupletted leaves and fracture crossing spanners:

```
>>> 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.

`MutationAgent.transpose(expr)`

Transposes notes and chords in mutation client by *expr*.

```
>>> staff = Staff()
>>> staff.append(Measure((4, 4), "c'4 d'4 e'4 r4"))
>>> staff.append(Measure((3, 4), "d'4 e'4 <f' a' c' '>4"))
>>> show(staff)
```



```
>>> mutate(staff).transpose("+m3")
>>> 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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

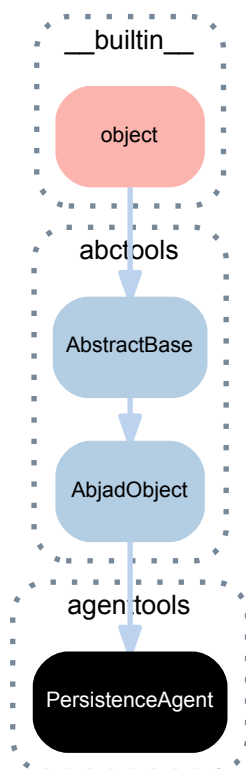
Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

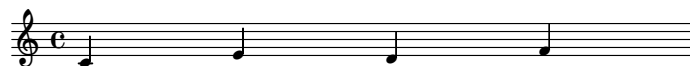
`(AbjadObject).__repr__()`
 Gets interpreter representation of Abjad object.
 Returns string.

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

Bases

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

Read-only properties

PersistenceAgent.**client**

Client of persistence agent.

Returns selection or component.

Methods

PersistenceAgent.**as_ly** (*ly_file_path=None*, *candidacy=False*, *illustrate_function=None*, ***kwargs*)

Persists client as LilyPond file.

Autogenerates file path when *ly_file_path* is none.

```
>>> 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
```

When *candidacy* is true, writes LilyPond output only when LilyPond output would differ from any existing output file.

When *candidacy* is false, writes LilyPond output even if LilyPond output would not differ from any existing output file.

Returns output path and elapsed formatting time when LilyPond output is written.

Returns false when when LilyPond output is not written due to *candidacy* being set.

PersistenceAgent.**as_midi** (*midi_file_path=None*, *remove_ly=False*, ***kwargs*)

Persists client as MIDI file.

Autogenerates file path when *midi_file_path* is none.

```
>>> 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_file_path*, *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_file_path=None*, *candidacy=False*, *illustrate_function=None*, *remove_ly=False*, ***kwargs*)

Persists client as PDF.

Autogenerates file path when *pdf_file_path* is none.

```
>>> 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
```

When *candidacy* is true, writes PDF output only when PDF output would differ from any existing output file.

When *candidacy* is false, writes PDF output even if PDF output would not differ from any existing output file.

Returns output path, elapsed formatting time and elapsed rendering time when PDF output is written.

Returns false when when PDF output is not written due to *candidacy* being set.

`PersistenceAgent.as_png(png_file_path=None, remove_ly=False, illustrate_function=None, **kwargs)`

Persists client as PNG.

```
>>> staff = Staff()
>>> measure = Measure((4, 4), "c'4 d'4 e'4 f'4")
>>> command = indicatortools.LilyPondCommand('break', 'after')
>>> attach(command, measure[-1])
>>> staff.extend(measure * 200)
```

```
>>> result = persist(staff).as_png()
>>> for x in result[0]:
...     x
...
'/Users/josiah/Desktop/test-page1.png'
'/Users/josiah/Desktop/test-page2.png'
'/Users/josiah/Desktop/test-page3.png'
'/Users/josiah/Desktop/test-page4.png'
'/Users/josiah/Desktop/test-page5.png'
'/Users/josiah/Desktop/test-page6.png'
'/Users/josiah/Desktop/test-page7.png'
'/Users/josiah/Desktop/test-page8.png'
'/Users/josiah/Desktop/test-page9.png'
'/Users/josiah/Desktop/test-page10.png'
'/Users/josiah/Desktop/test-page11.png'
'/Users/josiah/Desktop/test-page12.png'
'/Users/josiah/Desktop/test-page13.png'
'/Users/josiah/Desktop/test-page14.png'
'/Users/josiah/Desktop/test-page15.png'
```

Autogenerates file path when *png_file_path* is none.

Returns output path(s), 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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

Returns boolean.

(AbjadObject) .**__repr__**()

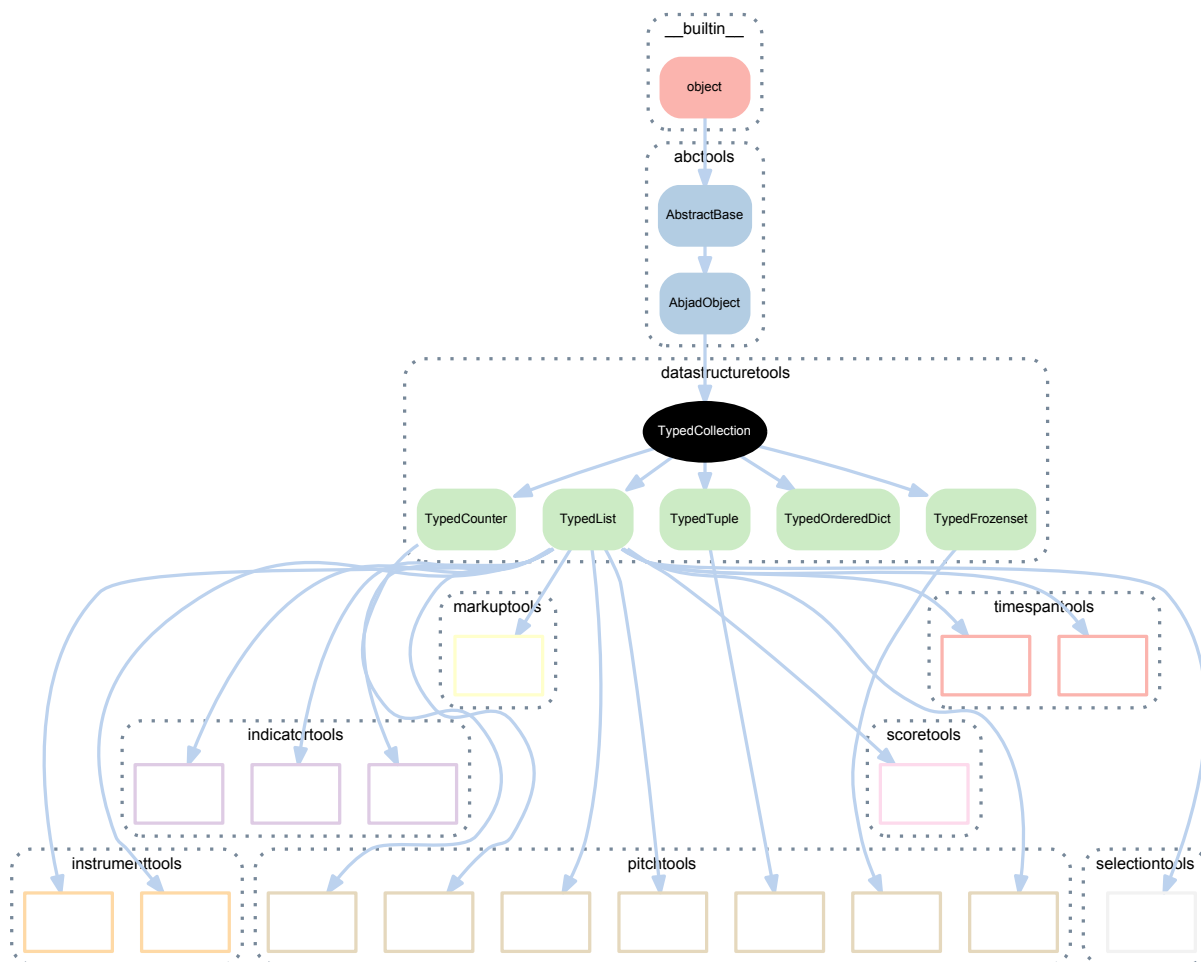
Gets interpreter representation of Abjad object.

Returns string.

DATASTRUCTURETOOLS

2.1 Abstract classes

2.1.1 datastructuretools.TypedCollection



class datastructuretools.**TypedCollection** (*items=None, item_class=None*)
Abstract base class for typed collections.

Bases

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

Read-only properties

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

`TypedCollection.items`
Gets collection items.

Special methods

`TypedCollection.__contains__(item)`
Is true when typed collection container *item*. Otherwise false.

Returns boolean.

`TypedCollection.__eq__(expr)`
Is 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.__hash__()`
Hashes typed collection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`TypedCollection.__iter__()`
Iterates typed collection.

Returns generator.

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

Returns nonnegative integer.

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

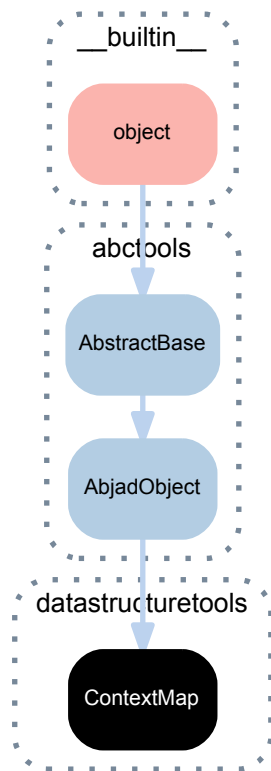
Returns boolean.

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

Returns string.

2.2 Concrete classes

2.2.1 datastructuretools.ContextMap



class datastructuretools.**ContextMap** (*score_template=None, settings=None*)

A context map.

```
>>> template = templatetools.StringOrchestraScoreTemplate()
>>> context_map = datastructuretools.ContextMap(template)
```

```
>>> context_map['String Orchestra Score']['color'] = 'red'
>>> context_map['Violin Staff Group']['color'] = 'blue'
>>> context_map['Contrabass Staff Group']['color'] = 'green'
>>> context_map['Contrabass 1 Voice']['color'] = 'yellow'
```

```
>>> print(format(context_map))
datastructuretools.ContextMap(
  score_template=templatetools.StringOrchestraScoreTemplate(
    violin_count=6,
    viola_count=4,
    cello_count=3,
    contrabass_count=2,
  ),
  settings=[
    (
      'Contrabass 1 Voice',
      [
        ('color', 'yellow'),
      ],
    ),
    (
      'Contrabass Staff Group',
      [
        ('color', 'green'),
      ],
    ),
    (
      'String Orchestra Score',
      [
```

```

        ('color', 'red'),
    ],
),
(
    'Violin Staff Group',
    [
        ('color', 'blue'),
    ],
),
],
)

```

```
>>> context_map['Violin 1 Voice']['color']
'blue'
```

```
>>> context_map['Viola 3 Voice']['color']
'red'
```

```
>>> context_map['Contrabass 1 Voice']['color']
'yellow'
```

```
>>> context_map['Contrabass 2 Voice']['color']
'green'
```

Bases

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

Read-only properties

`ContextMap.score_template`

Score template on which context map is based.

```
>>> print(format(context_map.score_template))
templatetools.StringOrchestraScoreTemplate(
    violin_count=6,
    viola_count=4,
    cello_count=3,
    contrabass_count=2,
)
```

Returns score template or none.

`ContextMap.settings`

All settings for context map.

Returns ordered dict or none.

Methods

`ContextMap.copy()`

Copy context map.

```
>>> copied_context_map = context_map.copy()
>>> print(format(copied_context_map))
datastructuretools.ContextMap(
    score_template=templatetools.StringOrchestraScoreTemplate(
        violin_count=6,
        viola_count=4,
        cello_count=3,
        contrabass_count=2,
    )
)
```

```

    ),
    settings=[
        (
            'Contrabass 1 Voice',
            [
                ('color', 'yellow'),
            ],
        ),
        (
            'Contrabass Staff Group',
            [
                ('color', 'green'),
            ],
        ),
        (
            'String Orchestra Score',
            [
                ('color', 'red'),
            ],
        ),
        (
            'Violin Staff Group',
            [
                ('color', 'blue'),
            ],
        ),
    ],
)

```

`ContextMap.update(expr)`

Updates context map with information in context map *expr*.

```

>>> template = templatetools.StringQuartetScoreTemplate()
>>> context_map_one = datastructuretools.ContextMap(template)
>>> context_map_one['Cello Voice']['one'] = 1
>>> context_map_one['Viola Voice']['foo'] = 'bar'
>>> context_map_two = datastructuretools.ContextMap(template)
>>> context_map_two['Cello Voice']['one'] = 'a'
>>> context_map_two['String Quartet Score']['baz'] = (1, 2, 3)

```

```

>>> print(format(context_map_one))
datastructuretools.ContextMap(
  score_template=templatetools.StringQuartetScoreTemplate(),
  settings=[
    (
      'Cello Voice',
      [
        ('one', 1),
      ],
    ),
    (
      'Viola Voice',
      [
        ('foo', 'bar'),
      ],
    ),
  ],
)

```

```

>>> print(format(context_map_two))
datastructuretools.ContextMap(
  score_template=templatetools.StringQuartetScoreTemplate(),
  settings=[
    (
      'Cello Voice',
      [
        ('one', 'a'),
      ],
    ),
    (
      'String Quartet Score',
      [

```

```
        (
            'baz',
            (1, 2, 3),
        ),
    ],
),
],
)
```

```
>>> context_map_one.update(context_map_two)
>>> print(format(context_map_one))
datastructuretools.ContextMap(
  score_template=templatetools.StringQuartetScoreTemplate(),
  settings=[
    (
      'Cello Voice',
      [
        ('one', 'a'),
      ],
    ),
    (
      'String Quartet Score',
      [
        (
          'baz',
          (1, 2, 3),
        ),
      ],
    ),
    (
      'Viola Voice',
      [
        ('foo', 'bar'),
      ],
    ),
  ],
)
```

Operates in place and 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 Abjad object.

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

Returns string.

ContextMap.**__getitem__**(*context_name*)

Gets context map component for *context_name*.

Returns context map component.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

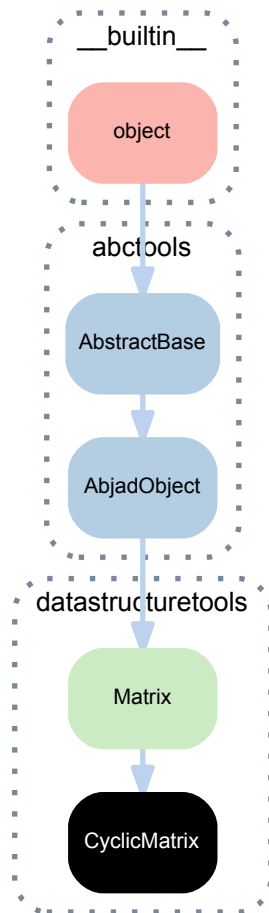
(AbjadObject).**__ne__**(*expr*)

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

Returns boolean.

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

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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad 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).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

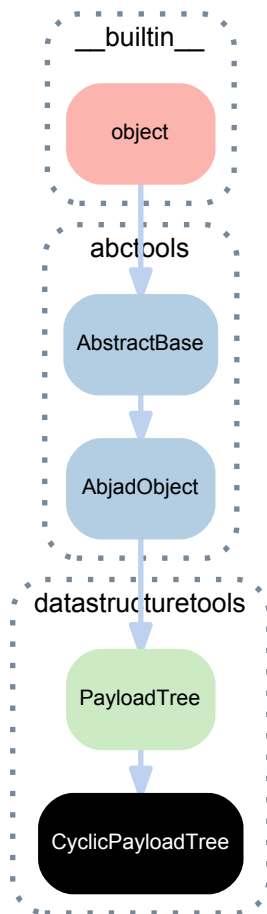
Returns boolean.

`CyclicMatrix.__repr__()`

Gets interpreter representation of cyclic matrix.

Returns string.

2.2.3 datastructuretools.CyclicPayloadTree



class datastructuretools.**CyclicPayloadTree** (*expr=None, item_class=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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`CyclicPayloadTree.children`

Children of cyclic payload tree.

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).expr`

Gets input argument.

`(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.

```

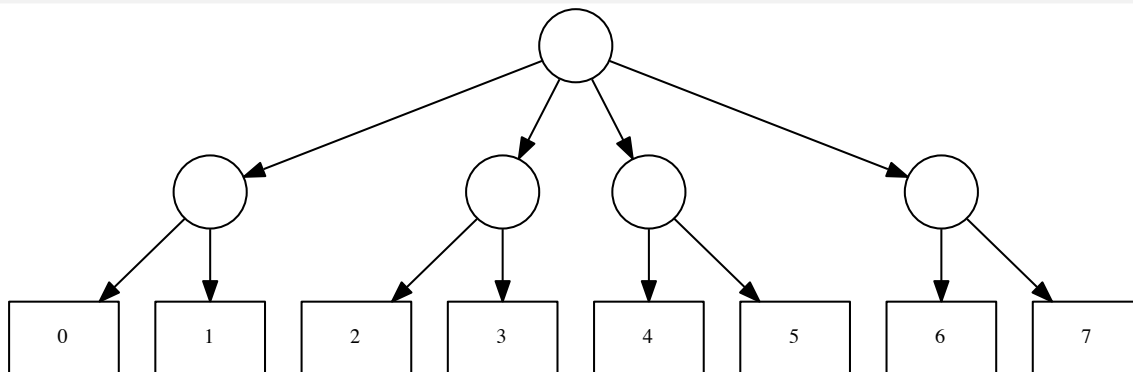
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> 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).**item_class**
Gets item class of payload tree.

```
>>> tree.item_class is None
True
```

Set item class to coerce input at initialization:

```
>>> tree = datastructuretools.PayloadTree(
...     expr=[[1.1, 2.2], [8.8, 9.9]],
...     item_class=int,
... )
>>> tree
PayloadTree([[1, 2], [8, 9]])
```

Returns class or none.

(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.

Payload tree of length greater than 1 for examples with positive *n*:

```
>>> 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])]
```

Payload tree of length greater than 1 for examples with negative n :

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

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 $level$.

Cyclic payload tree of length greater than 1 for examples with positive n :

```
>>> 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)
```

Cyclic payload tree of length greater than 1 for examples with negative n :

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

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)
```

Cyclic payload tree of length 1 for examples with positive n :

```
>>> sequence = [0]
>>> tree = datastructuretools.CyclicPayloadTree(sequence)
```

Gets next 4 nodes at level -1:

```
>>> for x in tree.get_next_n_nodes_at_level(4, -1):
...     x
CyclicPayloadTree(0)
CyclicPayloadTree(0)
CyclicPayloadTree(0)
CyclicPayloadTree(0)
```

Cyclic payload tree of length 1 for examples with negative n :

```
>>> sequence = [0]
>>> tree = datastructuretools.CyclicPayloadTree(sequence)
```

Gets next 4 nodes at level -1:

```
>>> for x in tree.get_next_n_nodes_at_level(-4, -1):
...     x
CyclicPayloadTree(0)
CyclicPayloadTree(0)
CyclicPayloadTree(0)
CyclicPayloadTree(0)
```

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*)
 Is 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.

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):
...     next(generator)
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):
...     next(generator)
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)`

Is 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)`

Is 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.

(PayloadTree).**__hash__**()

Hashes payload tree.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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.

Typical payload tree:

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

Payload tree leaf:

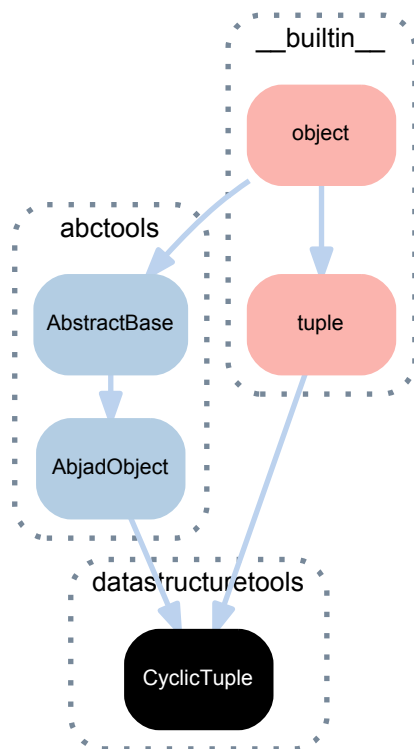
```
>>> datastructuretools.PayloadTree(0)
PayloadTree(0)
```

Empty payload tree:

```
>>> datastructuretools.PayloadTree()
PayloadTree([])
```

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`
- `abctools.AbjadObject.AbstractBase`
- `__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)
 Is true when *expr* is a tuple with items equal to those of this cyclic tuple. Otherwise false.
 Returns boolean.

(AbjadObject) .**__format__** (format_specification='')
 Formats Abjad 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.
 Gets slice open at right:

```
>>> sequence = [0, 1, 2, 3, 4, 5]
>>> sequence = datastructuretools.CyclicTuple(sequence)
>>> sequence[2:]
(2, 3, 4, 5)
```

Gets slice closed at right:

```
>>> sequence = [0, 1, 2, 3, 4, 5]
>>> sequence = datastructuretools.CyclicTuple(sequence)
>>> sequence[:15]
(0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2)
```

Returns tuple.

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

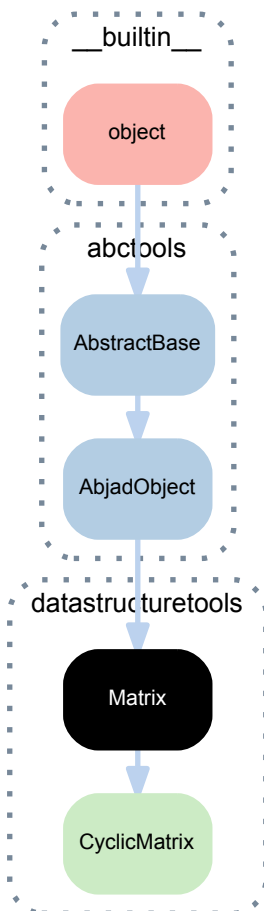
CyclicTuple .**__hash__** ()
 Hashes cyclic tuple.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 Returns integer.

(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
(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.
(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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad 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).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

Returns boolean.

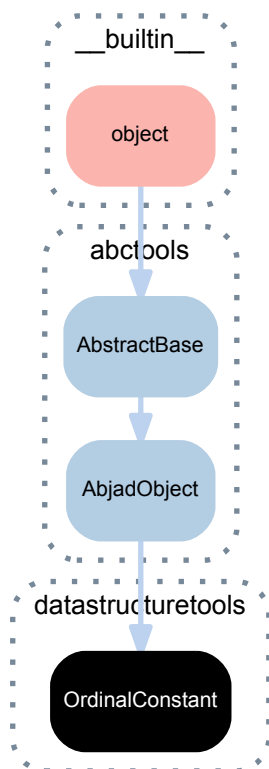
`Matrix.__repr__()`

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

Special methods

`OrdinalConstant.__eq__(expr)`

Is 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)`

$x._\text{ge}_\text{(y)} \iff x \geq y$

`OrdinalConstant.__gt__(other)`

$x._\text{gt}_\text{(y)} \iff x > y$

`OrdinalConstant.__hash__()`

Hashes ordinal constant.

Returns int.

`OrdinalConstant.__le__(other)`

$x._\text{le}_\text{(y)} \iff x \leq y$

`OrdinalConstant.__lt__(expr)`

Is 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.

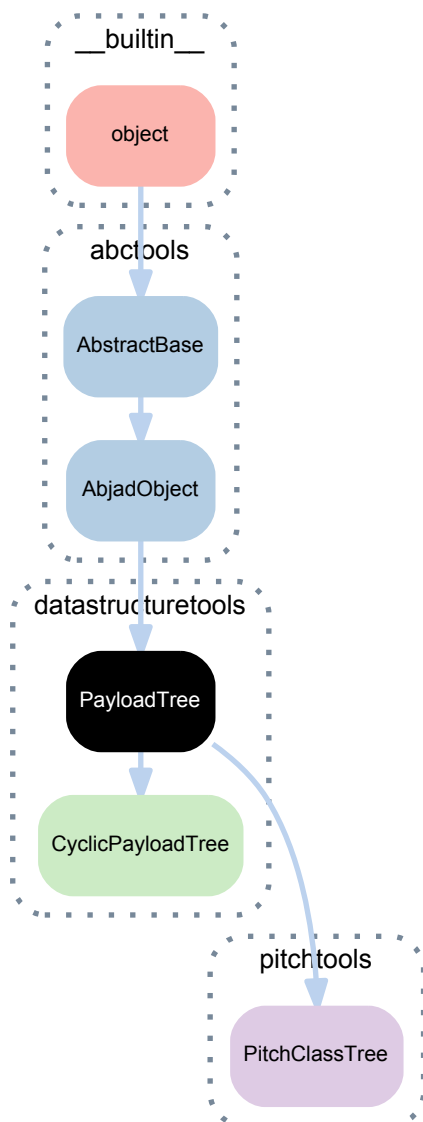
`OrdinalConstant.__new__(dimension=None, value=0, representation=None)`

`OrdinalConstant.__repr__()`

Gets interpreter representation of ordinal constant.

Returns string.

2.2.7 datastructuretools.PayloadTree



class datastructuretools.**PayloadTree** (*expr=None, item_class=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`
- `abctools.AbjadObject.AbstractBase`
- `__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.expr
Gets input argument.

PayloadTree.graphviz_format
Graphviz format of payload tree.

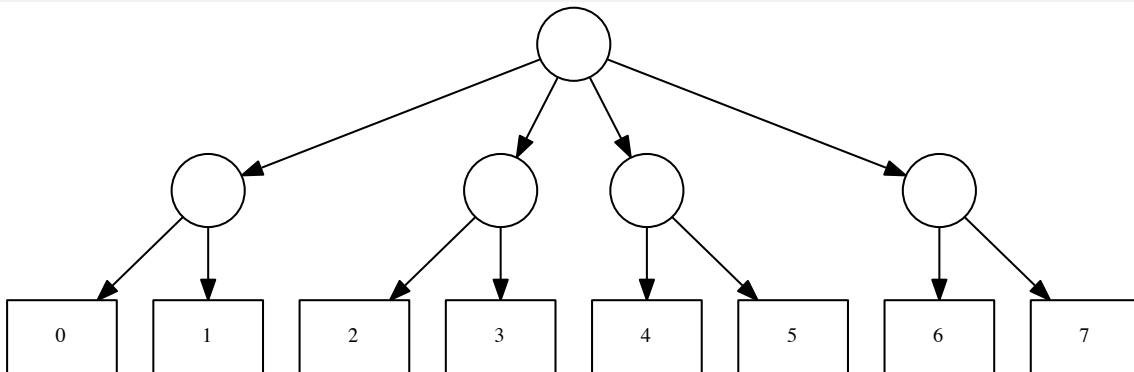
```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)
>>> print(tree.graphviz_format)
digraph G {
    node_0 [label="",
            shape=circle];
    node_1 [label="",
            shape=circle];
    node_2 [label=0,
            shape=box];
    node_3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

Gets item class of payload tree.

```
>>> tree.item_class is None
True
```

Set item class to coerce input at initialization:

```
>>> tree = datastructuretools.PayloadTree(
...     expr=[[1.1, 2.2], [8.8, 9.9]],
...     item_class=int,
... )
>>> tree
PayloadTree([[1, 2], [8, 9]])
```

Returns class or none.

`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.

Payload tree of length greater than 1 for examples with positive *n*:

```
>>> 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])]
```

Payload tree of length greater than 1 for examples with negative *n*:

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

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*)
 Is 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)`
Is 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)`
Is 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.__hash__()`
Hashes payload tree.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`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.

Typical payload tree:

```

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

```

Payload tree leaf:

```

>>> datastructuretools.PayloadTree(0)
PayloadTree(0)

```

Empty payload tree:

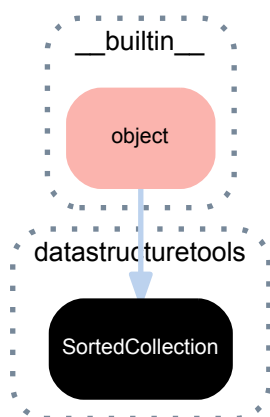
```

>>> datastructuretools.PayloadTree()
PayloadTree([])

```

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)`
 Is true when sorted collection contains *item*. Otherwise false.
 Returns boolean.

`SortedCollection.__getitem__(i)`
 Gets *i* in sorted collection.
 Returns item.

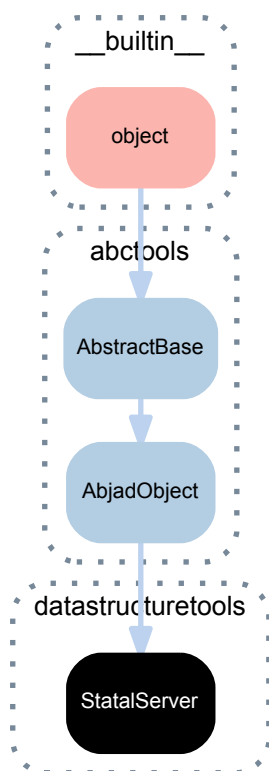
`SortedCollection.__iter__()`
 Iterates sorted collection.
 Yields items.

`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`
- `abctools.AbjadObject.AbstractBase`
- `__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*)
Is 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 Abjad object.

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

Returns string.

`StatalServer.__hash__` ()
Hashes statal server.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

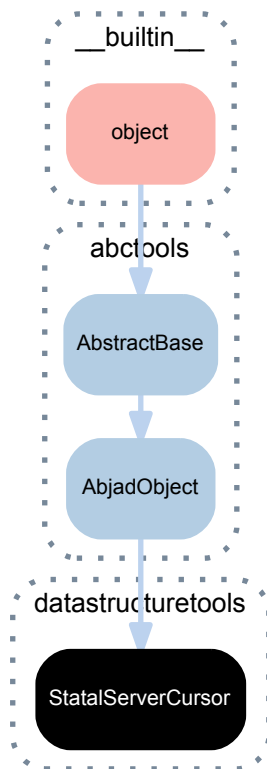
`(AbjadObject).__ne__` (*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

Returns string.

2.2.10 `datastructuretools.StatalServerCursor`



class `datastructuretools.StatalServerCursor` (*statal_server=None, position=None, reverse=False*)

A statal server cursor.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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. Is 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)`

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

Gets manifest payload of nodes at level -1:

```
>>> sequence = [(0, 1), (2, 3), (4, 5), (6, 7)]
>>> server = datastructuretools.StatalServer(sequence)
>>> cursor = server()
```

```
>>> cursor()
[0]
>>> cursor()
[1]
>>> cursor()
[2]
>>> cursor()
[3]
>>> cursor()
[4]
>>> cursor()
[5]
>>> cursor()
[6]
>>> cursor()
[7]
>>> cursor()
[0]
>>> cursor()
[1]
```

Gets manifest payload of nodes at level -2:

```
>>> sequence = [(0, 1), (2, 3), (4, 5), (6, 7)]
>>> server = datastructuretools.StatalServer(sequence)
>>> cursor = server()
```

```
>>> cursor(level=-2)
[0, 1]
>>> cursor(level=-2)
[2, 3]
>>> cursor(level=-2)
[4, 5]
>>> cursor(level=-2)
[6, 7]
>>> cursor(level=-2)
[0, 1]
```

Gets manifest payload of nodes at level -1 for statal server of length 1:

```
>>> sequence = [0]
>>> server = datastructuretools.StatalServer(sequence)
>>> cursor = server()
```

```
>>> cursor()
[0]
>>> cursor()
[0]
>>> cursor()
[0]
>>> cursor()
[0]
>>> cursor()
[0]
```

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 Abjad object.

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

Returns string.

StatalServerCursor.**__hash__**()

Hashes statal server cursor.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

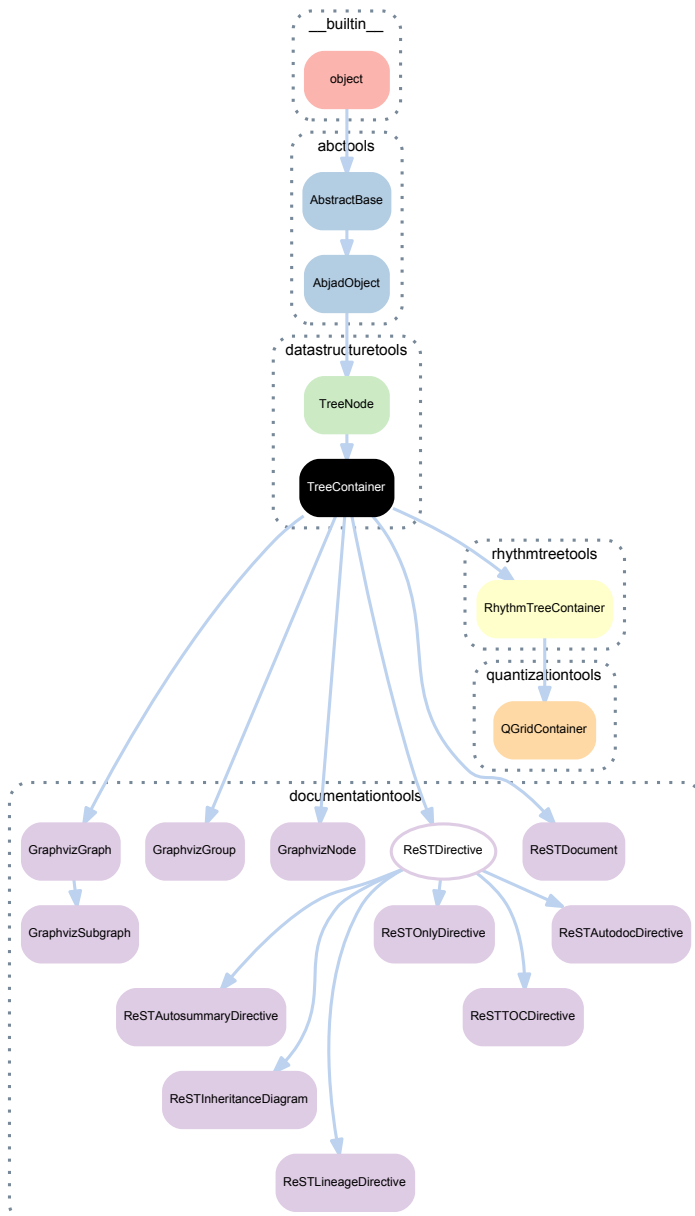
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad 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.__hash__()`
Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`TreeContainer.__iter__()`
Iterates tree container.

Yields children of tree container.

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

`(TreeNode).__ne__(expr)`
Is 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
```

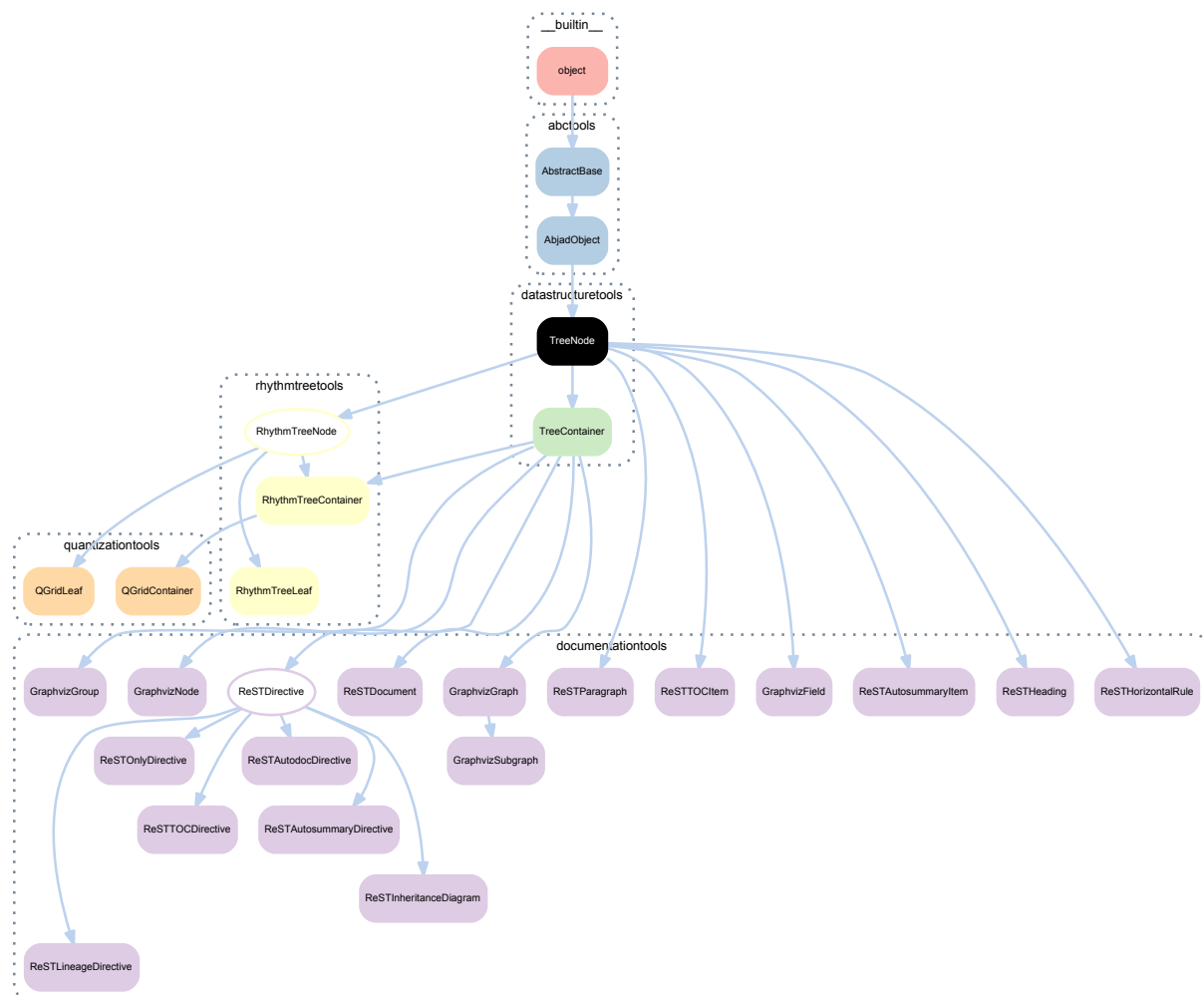
```
>>> c.parent is a
True
```

```
>>> 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is true when *expr* is a tree node. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`TreeNode.__hash__()`

Hashes tree node.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`TreeNode.__ne__(expr)`

Is true when tree node does not equal *expr*. Otherwise false.

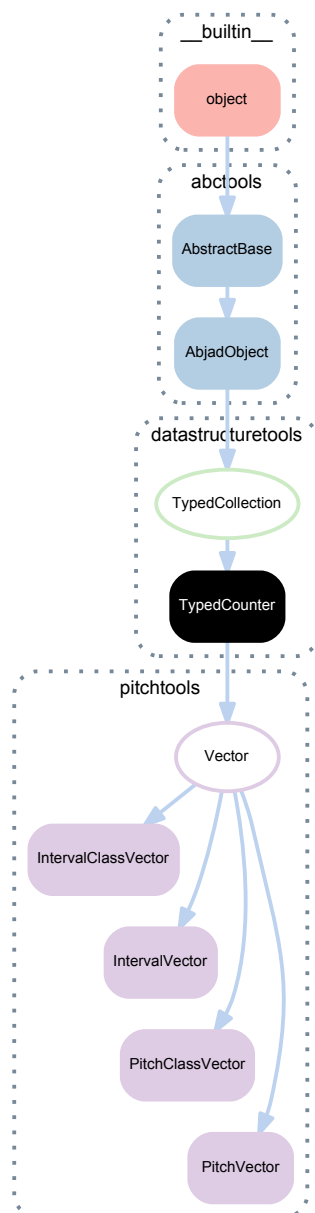
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

2.2.13 datastructuretools.TypedCounter



class datastructuretools.**TypedCounter** (*items=None, item_class=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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

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()`
Iterates items in typed counter.

Yields items.

`TypedCounter.keys()`
Iterates 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()`
Iterates values in 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__(item)`

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

`TypedCounter.__delitem__(item)`

Deletes *item* from typed counter.

Returns none.

`(TypedCollection).__eq__(expr)`

Is 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__(item)`

Gets *item* from typed counter.

Returns item.

`(TypedCollection).__hash__()`

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(TypedCollection).__iter__()`

Iterates typed collection.

Returns generator.

`(TypedCollection).__len__()`

Length of typed collection.

Returns nonnegative integer.

`TypedCounter.__missing__(item)`

Returns zero.

Returns zero.

`(TypedCollection).__ne__(expr)`

Is 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__(item, value)`

Sets typed counter *item* to *value*.

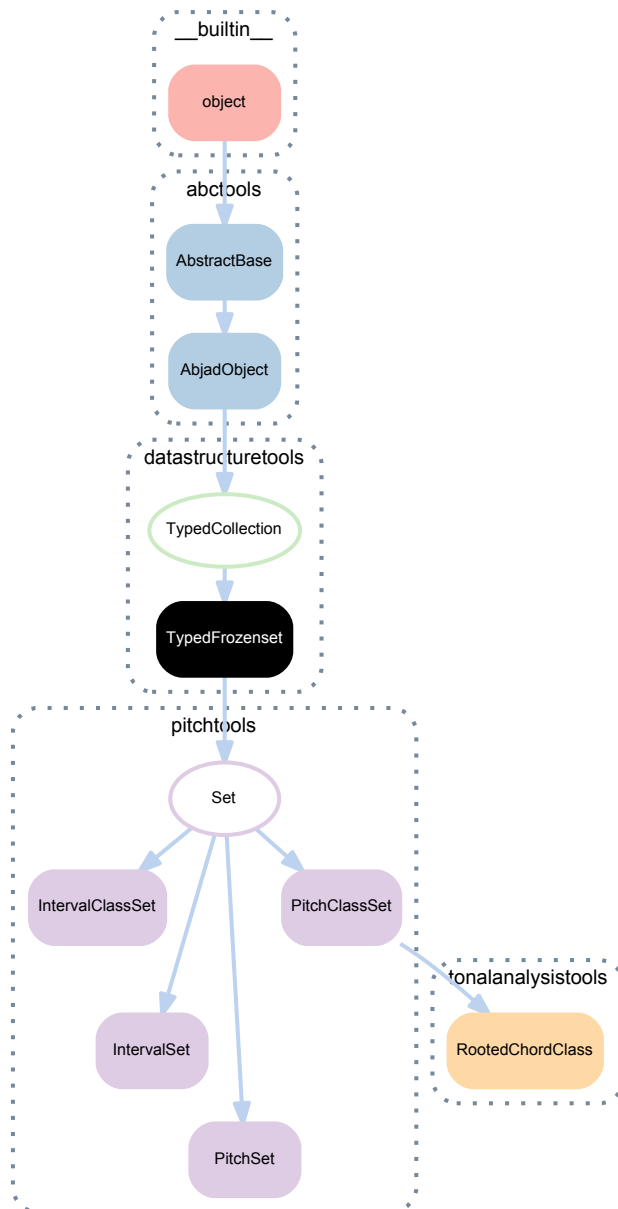
Returns none.

`TypedCounter.__sub__(expr)`

Subtracts *expr* from typed counter.

Returns new typed counter.

2.2.14 datastructuretools.TypedFrozenset



class `datastructuretools.TypedFrozenset` (*items=None, item_class=None*)

A typed frozen set.

Bases

- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

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)`
Is true when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

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

Returns boolean.

`TypedFrozenSet.issuperset(expr)`
Is 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__(item)`
Is true when typed collection container *item*. Otherwise false.

Returns boolean.

`(TypedCollection).__eq__(expr)`
 Is 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)`
 Is true when typed frozen set is greater than or equal to *expr*. Otherwise false.
 Returns boolean.

`TypedFrozenSet.__gt__(expr)`
 Is 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)`
 Is 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)`
 Is true when typed frozen set is less than *expr*. Otherwise false.
 Returns boolean.

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

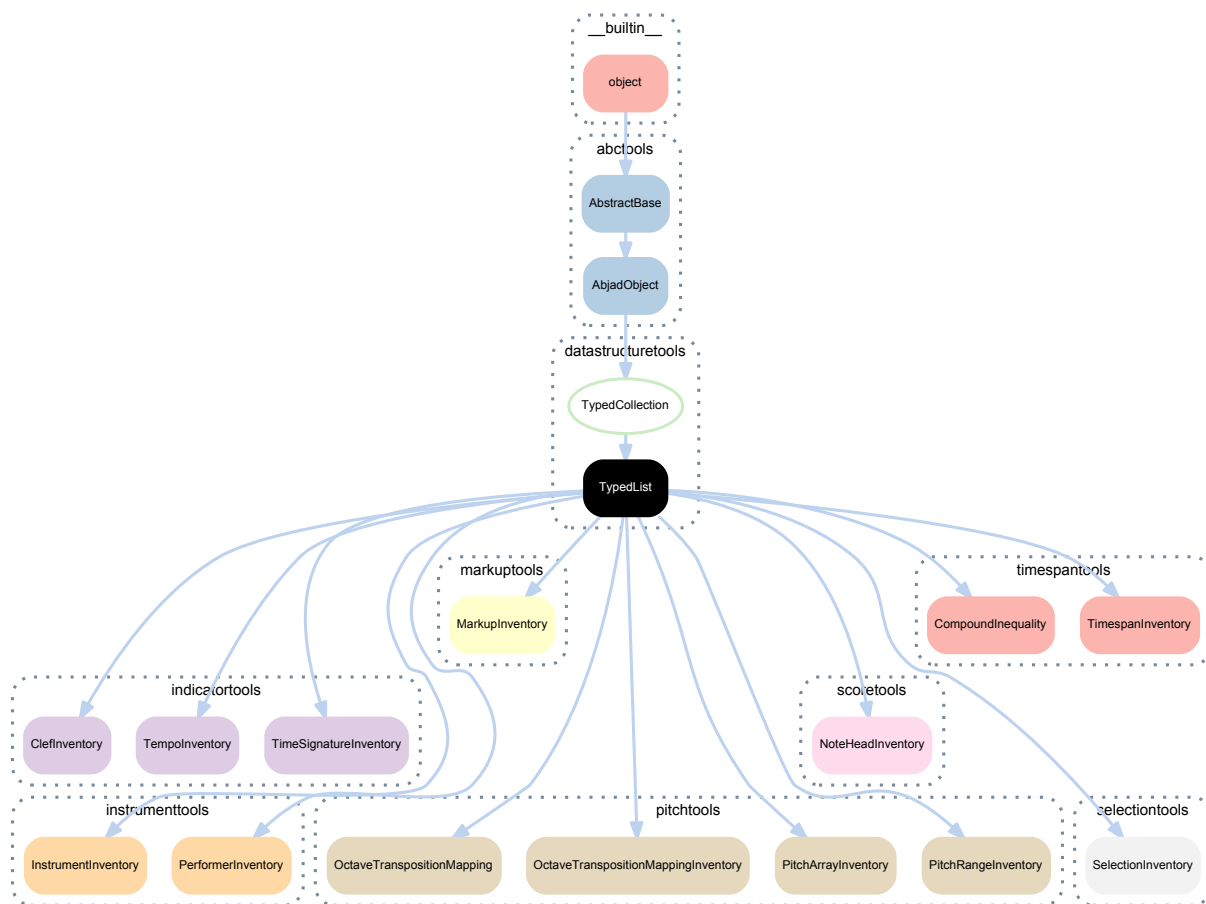
`TypedFrozenSet.__or__(expr)`
 Logical OR of typed frozen set and *expr*.
 Returns new typed frozen set.

`(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** (*items=None, item_class=None, keep_sorted=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,
])
```

```
>>> 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`TypedList.append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`TypedList.extend(items)`

Changes *items* 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(item)`

Changes *item* to item and returns index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     items=('cqu', 'as', 'b', 'dss'),
...     item_class=NamedPitch)
>>> pitch_collection.index("as")
1
```

Returns index.

`TypedList.insert(i, item)`

Changes *item* 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(item)`

Changes *item* 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__(item)`

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

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

Returns none.

`(TypedCollection).__eq__(expr)`
 Is 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__()`.

Returns item.

`(TypedCollection).__hash__()`
 Hashes typed collection.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`TypedList.__iadd__(expr)`
 Changes items 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(
        name='ppp',
    ),
    indicatortools.Dynamic(
        name='p',
    ),
    indicatortools.Dynamic(
        name='mp',
    ),
    indicatortools.Dynamic(
        name='mf',
    ),
    indicatortools.Dynamic(
        name='fff',
    ),
],
item_class=indicatortools.Dynamic,
)
```

Returns collection.

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

Returns generator.

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

Returns nonnegative integer.

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

Is 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__**().

Returns generator.

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

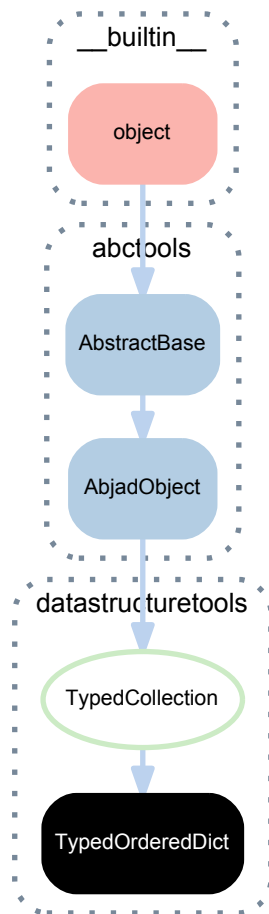
Changes items in *expr* to items and sets.

```
>>> 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,
)
```

Returns none.

2.2.16 datastructuretools.TypedOrderedDict



class datastructuretools.**TypedOrderedDict** (*items=None, item_class=None*)
 A typed ordered dictionary.

Note: Doctests not included because class is used as a base class. Doctests here would put weird examples in child classes. See child classes for doctests.

Bases

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

Read-only properties

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

Methods

TypedOrderedDict.**clear**()
 Clears typed ordered dictionary.

Returns none.

`TypedOrderedDict.copy()`
Copies typed ordered dictionary.

Returns new typed ordered dictionary.

`TypedOrderedDict.get(i, default=None)`
Aliases `OrderedDict.get()`.

Returns item or raises key error.

`TypedOrderedDict.has_key(key)`
Aliases `OrderedDict.has_key()`.

Returns boolean.

`TypedOrderedDict.items()`
Aliases `OrderedDict.items()`.

Returns generator.

`TypedOrderedDict.keys()`
Aliases `OrderedDict.keys()`.

Returns generator.

`TypedOrderedDict.pop(key, default=None)`
Aliases `OrderedDict.pop()`.

Returns items.

`TypedOrderedDict.popitem()`
Aliases `OrderedDict.popitem()`.

Returns generator.

`TypedOrderedDict.setdefault(key, default=None)`
Aliases `OrderedDict.setdefault()`.

Returns items.

`TypedOrderedDict.update(*args, **kwargs)`
Aliases `OrderedDict.update()`.

Returns none.

`TypedOrderedDict.values()`
Aliases `OrderedDict.values()`.

Returns generator.

Special methods

`TypedOrderedDict.__cmp__(expr)`
Aliases `OrderedDict.__cmp__()`.

Returns boolean.

`TypedOrderedDict.__contains__(key)`
Aliases `OrderedDict.__contains__()`.

Returns boolean.

`TypedOrderedDict.__delitem__(i)`
Aliases `OrderedDict.__delitem__()`.

Returns none.

(TypedCollection).**__eq__**(*expr*)

Is 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.

TypedOrderedDict.**__ge__**(*expr*)

Is true when typed ordered dictionary is greater than or equal to *expr*. Otherwise false.

Returns boolean.

TypedOrderedDict.**__getitem__**(*i*)

Aliases OrderedDict.**__getitem__**().

Returns item.

TypedOrderedDict.**__gt__**(*expr*)

Is true when typed ordered dictionary is greater than *expr*. Otherwise false.

Returns boolean.

(TypedCollection).**__hash__**()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

TypedOrderedDict.**__le__**(*expr*)

Is true when typed ordered dictionary is less than or equal to *expr*. Otherwise false.

Returns boolean.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

TypedOrderedDict.**__lt__**(*expr*)

Is true when typed ordered dictionary is less than *expr*. Otherwise false.

Returns boolean.

TypedOrderedDict.**__ne__**(*expr*)

Is true when typed ordered dictionary is not equal to *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

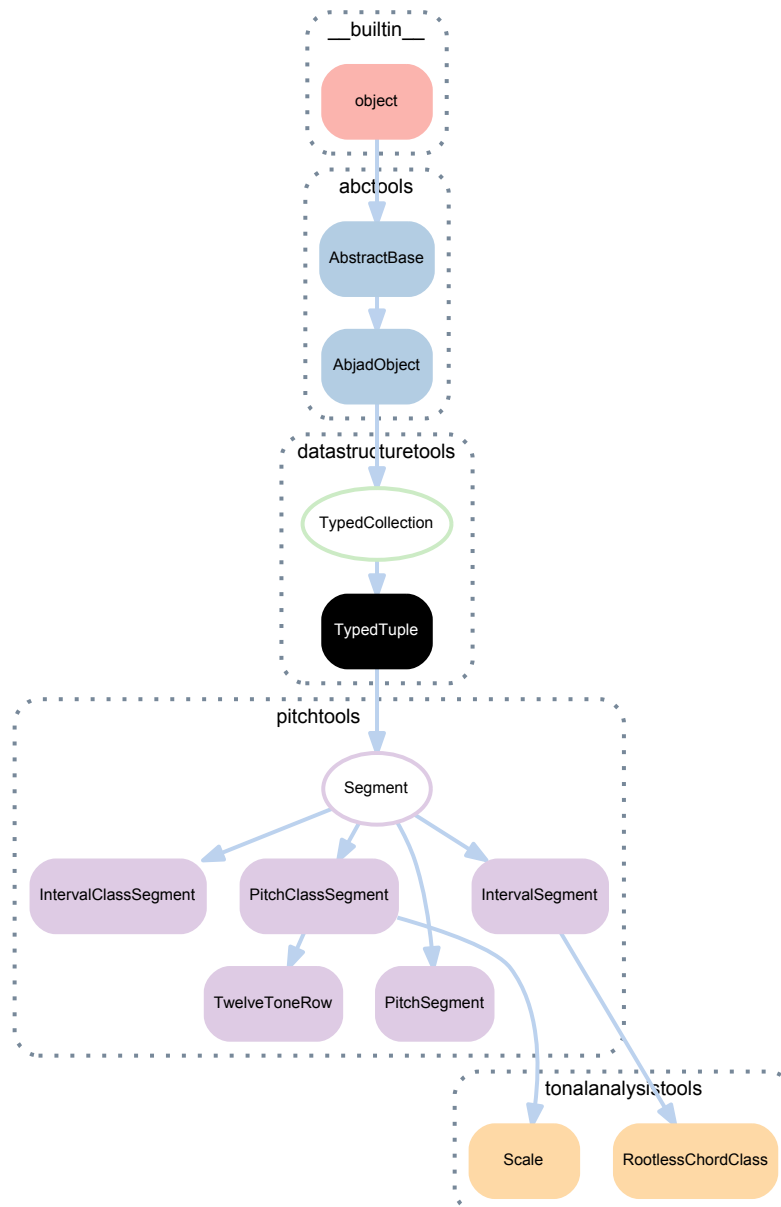
TypedOrderedDict.**__reversed__**()

Aliases OrderedDict.**__reversed__**().

Returns generators.

`TypedOrderedDict.__setitem__(i, expr)`
 Changes items in *expr* to items and sets.
 Returns none.

2.2.17 datastructuretools.TypedTuple



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

Bases

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Methods

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

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

Special methods

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

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

`(TypedCollection).__eq__(expr)`
Is 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.

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

Multiplies typed tuple by *expr*.

Returns new typed tuple.

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

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

Returns boolean.

TypedTuple .**__radd__**(*expr*)

Right-adds *expr* to typed tuple.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

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

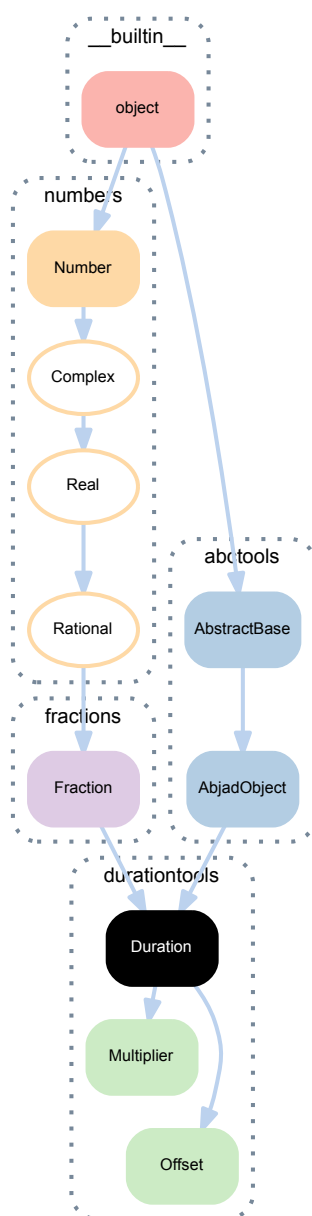
Multiplies *expr* by typed tuple.

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)
```

Initializes 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(6, 32))
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`
- `abctools.AbjadObject.AbstractBase`

- `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

Is 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 multipler.

Duration.**is_assignable**

Is 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.

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 = next(generator)
...     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*)

Is 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):
...     next(generator)
...
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):
...     next(generator)
...
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)`

Is 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)`

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

Returns boolean.

`Duration.__gt__(arg)`

Is true when duration is greater than *arg*. Otherwise false.

Returns boolean.

`Duration.__hash__()`

Hashes duration.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`Duration.__le__(arg)`

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

Returns boolean.

`Duration.__lt__(arg)`

Is 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)`

Is 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.

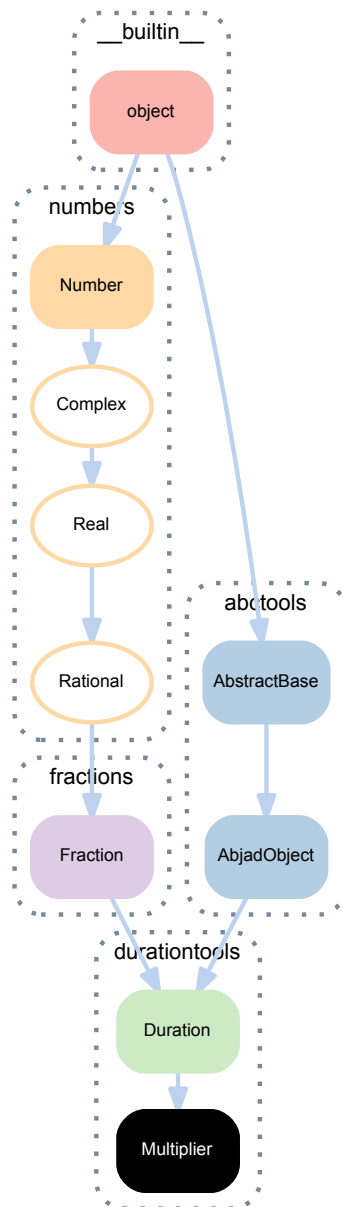
`Duration.__truediv__(*args)`

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`
- `abctools.AbjadObject.AbstractBase`
- `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**

Is 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 multipler.

(Duration).**.is_assignable**

Is 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.

Multiplier.is_proper_tuplet_multiplier

Is 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 = next(generator)
...     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 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*)

Is 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):
...     next(generator)
...
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):
...     next(generator)
...
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)
Is 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)
Is true when duration is greater than or equal to arg. Otherwise false.
```

Returns boolean.

```
(Duration) .__gt__(arg)
Is true when duration is greater than arg. Otherwise false.
```

Returns boolean.

```
(Duration) .__hash__()
Hashes duration.
```

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(Duration) .__le__(arg)
Is true when duration is less than or equal to *arg*. Otherwise false.
Returns boolean.

(Duration) .__lt__(arg)
Is 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)
Is 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) .__**rtuediv**__ (*args)

Documentation required.

Returns new duration.

(Fraction) .__**str**__ ()

str(self)

(Duration) .__**sub**__ (*args)

Subtracts *args* from duration.

Returns new duration.

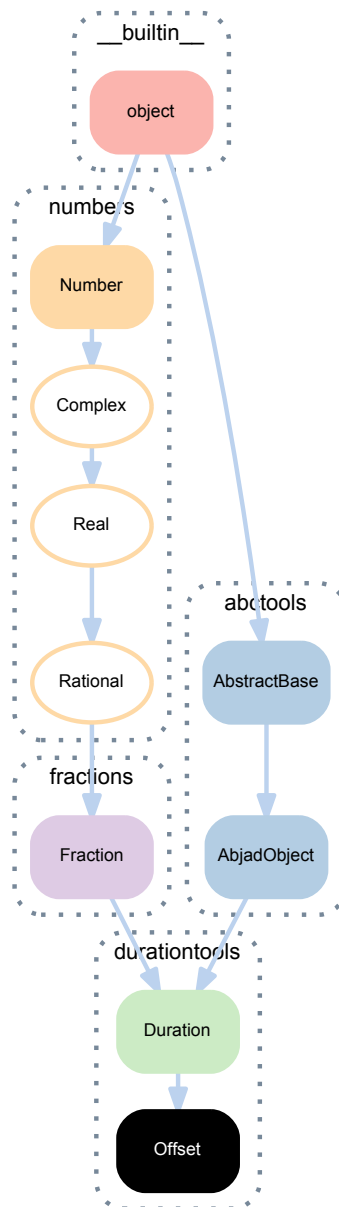
(Duration) .__**tuediv**__ (*args)

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`
- `abctools.AbjadObject.AbstractBase`
- `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{!t}'
...         string = string.format(sixteenths, dot_count)
...         print(string)
...     except AssignabilityError:
...         sixteenths = duration.with_denominator(16)
...         string = '{!s}\t{!t}'
...         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{!t}\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{!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**

Is 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{!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**

Is 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.

(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 = next(generator)
...     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 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)`
 Is 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):
...     next(generator)
...
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):
...     next(generator)
...
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)
Is 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)  
Is true when duration is greater than or equal to arg. Otherwise false.  
  
Returns boolean.
```

```
(Duration) .__gt__(arg)  
Is true when duration is greater than arg. Otherwise false.  
  
Returns boolean.
```

```
(Duration) .__hash__()  
Hashes duration.  
  
Required to be explicitly re-defined on Python 3 if __eq__ changes.  
  
Returns integer.
```

```
(Duration) .__le__(arg)  
Is true when duration is less than or equal to arg. Otherwise false.  
  
Returns boolean.
```

```
(Duration) .__lt__(arg)  
Is 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)  
Is 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)

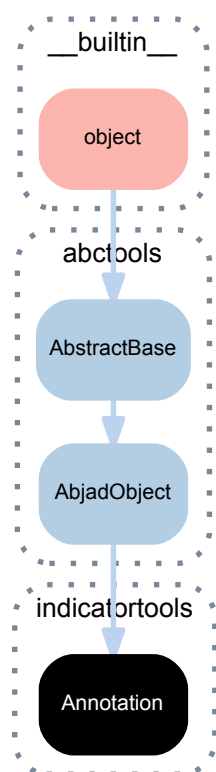
Documentation required.

(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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`Annotation.__hash__()`

Hashes annotation.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

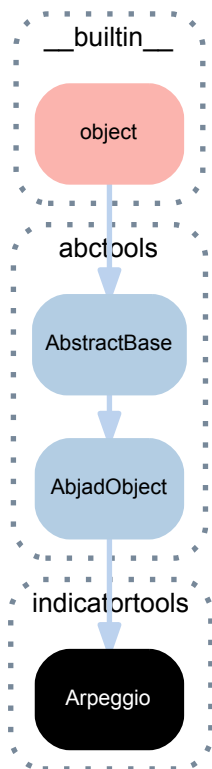
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.2 indicatortools.Arpeggio



class `indicatortools.Arpeggio` (*direction=None*)
 An arpeggio indication.

```
>>> chord = Chord("<c' e' g' c''>4")
>>> arpeggio = indicatortools.Arpeggio()
>>> attach(arpeggio, chord)
>>> show(chord)
```



An arpeggio arrow direction can be specified:

```
>>> chord = Chord("<c' e' g' c''>4")
>>> arpeggio = indicatortools.Arpeggio(direction=Down)
>>> attach(arpeggio, chord)
>>> show(chord)
```



Bases

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

Read-only properties

`Arpeggio.direction`
 Gets arpeggio arrow direction.

Return ordinal constant or none.

Special methods

`Arpeggio.__eq__(expr)`

Is true when *expr* is an arpeggio indication with a direction equal to that of this arpeggio indication. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`Arpeggio.__hash__()`

Hashes arpeggio.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

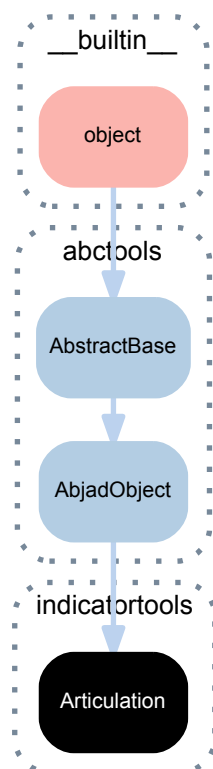
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.3 indicatortools.Arteulation



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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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.__hash__()`

Hashes articulation.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`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.

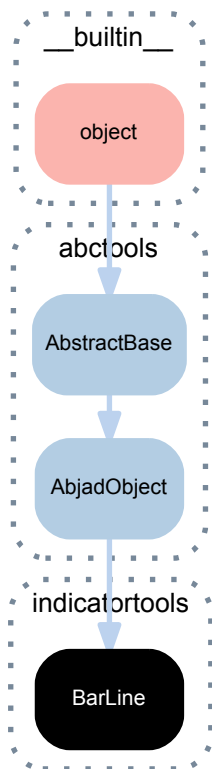
Returns string.

`Articulation.__str__()`

String representation of articulation.

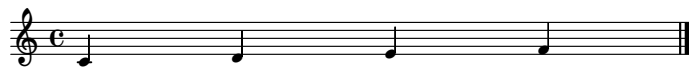
Returns string.

4.1.4 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`BarLine.__hash__()`

Hashes bar line.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

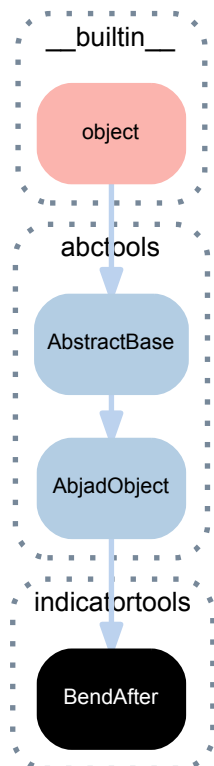
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.5 indicatortools.BendAfter



class `indicatortools.BendAfter` (*bend_amount=-4*)
 A fall or doit.

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



Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)
 Is true when *expr* is a bend after indication with bend amount equal to that of this bend after indication after.
 Otherwise false.

Returns boolean.

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

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

Returns string.

`BendAfter.__hash__` ()
 Hashes bend after.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

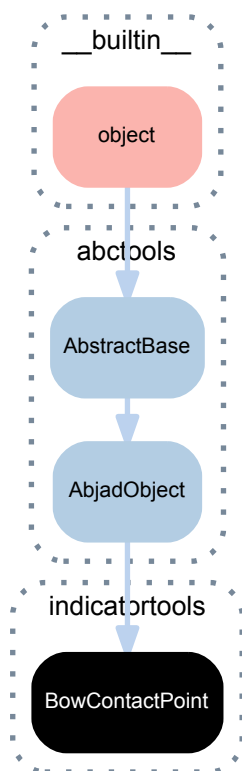
Returns boolean.

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

Returns string.

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

4.1.6 `indicatortools.BowContactPoint`



class `indicatortools.BowContactPoint` (*contact_point=None*)
 Bow contact point indicator.
 Contact points are measured from frog to tip as a fraction between 0 and 1.

```

>>> indicator = indicatortools.BowContactPoint((1, 2))
>>> print(format(indicator))
indicatortools.BowContactPoint(
    contact_point=durationtools.Multiplier(1, 2),
)
    
```

Bases

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

Read-only properties

`BowContactPoint.contact_point`
 Gets contact point.

```

>>> indicator = indicatortools.BowContactPoint((1, 4))
>>> indicator.contact_point
Multiplier(1, 4)
    
```

Returns multiplier.

`BowContactPoint.markup`

Gets bow contact point markup.

```
>>> indicator = indicatortools.BowContactPoint((3, 4))
>>> print(format(indicator.markup, 'lilypond'))
\markup {
  \vcenter
    \fraction
      3
      4
}
```

Special methods

`BowContactPoint.__eq__(expr)`

Is true if *expr* is a bow contact point with the same contact point as this bow contact point.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`BowContactPoint.__ge__(other)`

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

`BowContactPoint.__gt__(other)`

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

`BowContactPoint.__hash__()`

Hashes bow contact point.

`BowContactPoint.__le__(other)`

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

`BowContactPoint.__lt__(expr)`

Is true if *expr* is a bow contact point and this bow contact point is less than *expr*.

Returns boolean.

`(AbjadObject).__ne__(expr)`

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

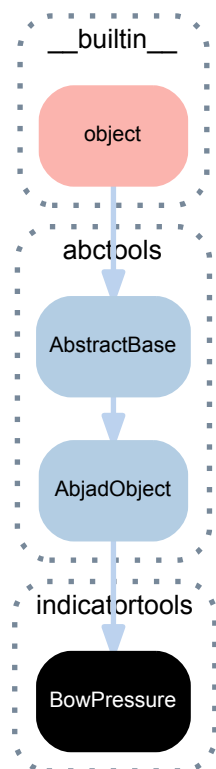
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.7 indicatortools.BowPressure



class `indicatortools.BowPressure` (*pressure=None*)
 Bow pressure indicator.

```

>>> indicator = indicatortools.BowPressure('overpressure')
>>> print(format(indicator))
indicatortools.BowPressure(
    pressure='overpressure',
)
  
```

Bases

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

Read-only properties

`BowPressure.pressure`
 Gets pressure.

```

>>> indicator = indicatortools.BowPressure('underpressure')
>>> indicator.pressure
'underpressure'
  
```

Special methods

`(AbjadObject).__eq__(expr)`
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

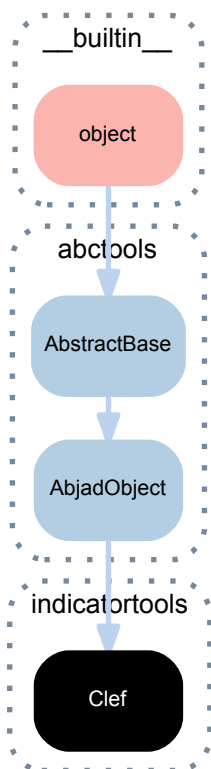
(AbjadObject).**__format__**(*format_specification*='')
 Formats Abjad object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject).**__hash__**()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 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.

4.1.8 indicatortools.Clef



class `indicatortools.Clef` (*name*='treble')
 A clef.

```
>>> clef = Clef('treble')
>>> clef
Clef(name='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`
- `abctools.AbjadObject.AbstractBase`
- `__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(name='alto'), Clef(name='alto'))
```

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

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

Returns new clef.

`Clef.__eq__(expr)`

Is true when *expr* is a clef with name equal to that of this 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(
    name='treble',
)
```

Returns string.

`Clef.__hash__()`

Hashes clef.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`Clef.__ne__(arg)`

Is 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
```

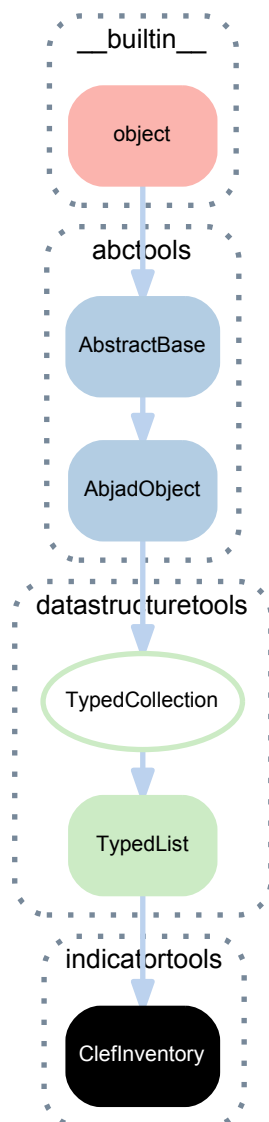
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.9 indicatortools.ClefInventory



class `indicatortools.ClefInventory` (*items=None, item_class=None, keep_sorted=None*)
 An ordered list of clefs.

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

```
>>> inventory
ClefInventory([Clef(name='treble'), Clef(name='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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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** (*item*)

Changes *item* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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__** (*item*)

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

Returns none.

(TypedCollection).**__eq__**(*expr*)

Is 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__**().

Returns item.

(TypedCollection).**__hash__**()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedList).**__iadd__**(*expr*)

Changes items in *expr* to items and extends.

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

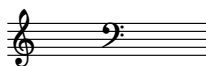
```
>>> print(format(dynamic_collection))
datastructuretools.TypedList(
[
    indicatortools.Dynamic(
        name='ppp',
    ),
    indicatortools.Dynamic(
        name='p',
    ),
    indicatortools.Dynamic(
        name='mp',
    ),
    indicatortools.Dynamic(
        name='mf',
    ),
    indicatortools.Dynamic(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

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

Is 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__**().

Returns generator.

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

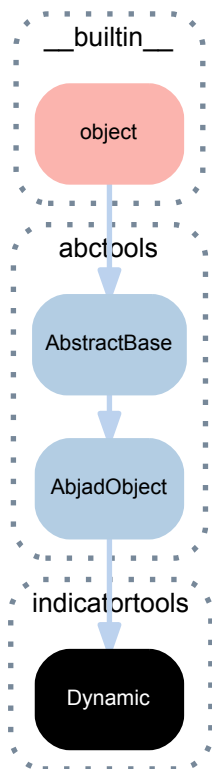
Changes items in *expr* to items and sets.

```
>>> 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,
)
```

Returns none.

4.1.10 `indicatortools.Dynamic`



class `indicatortools.Dynamic` (*name='f'*)
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(name='f')
```

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

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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__(expr)`

Is true when *expr* is a dynamic with a name equal to that of this dynamic. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`Dynamic.__hash__()`

Hashes dynamic.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

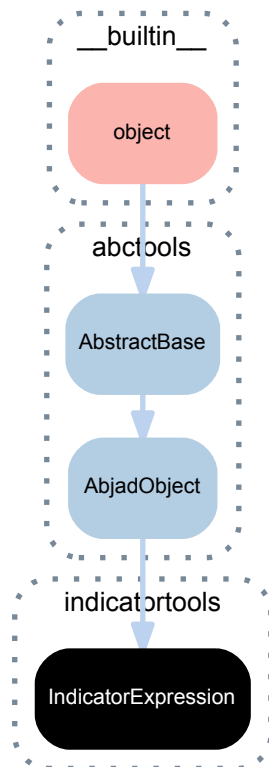
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

4.1.11 indicatortools.IndicatorExpression



class `indicatortools.IndicatorExpression` (*indicator=None*, *scope=None*, *component=None*)
 An indicator expression.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`IndicatorExpression.__hash__()`

Hashes indicator expression.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

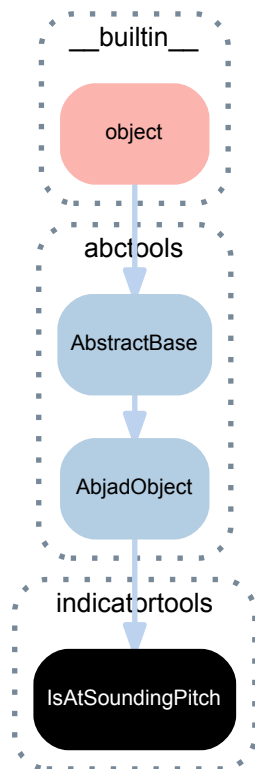
Returns boolean.

`IndicatorExpression.__repr__()`

Gets interpreter representation of indicator expression.

Returns string.

4.1.12 `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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

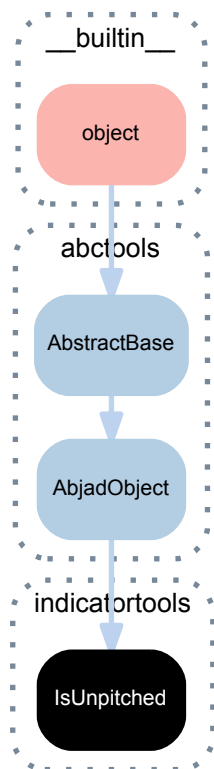
Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

4.1.13 indicatortools.IsUnpitched



class `indicatortools.IsUnpitched`
 Is unpitched indicator.

```
>>> indicator = indicatortools.IsUnpitched()
```

Attach to score selection to denote unpitched music.

Bases

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

Special methods

(AbjadObject).**__eq__**(*expr*)
 Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
 Returns boolean.

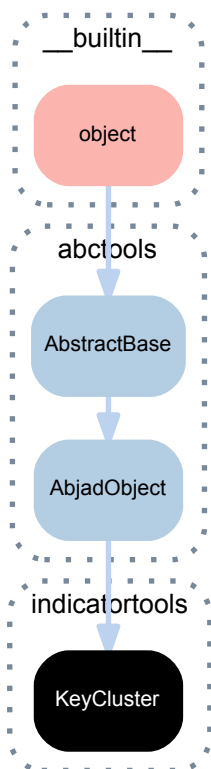
(AbjadObject).**__format__**(*format_specification*='')
 Formats Abjad object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject).**__hash__**()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 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.

4.1.14 `indicatortools.KeyCluster`



class `indicatortools.KeyCluster` (*include_black_keys=True*, *include_white_keys=True*,
markup_direction=None, *suppress_markup=False*)

A key cluster indication.

```

>>> chord = Chord("<c' e' g' b' d'' f''>8")
>>> key_cluster = indicatortools.KeyCluster()
>>> attach(key_cluster, chord)
>>> show(chord)
  
```



Bases

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

Read-only properties

`KeyCluster.include_black_keys`

Is true if key cluster includes black keys.

Returns boolean.

`KeyCluster.include_white_keys`

Is true if key cluster includes white keys.

Returns boolean.

`KeyCluster.markup_direction`

Gets markup direction.

Returns ordinal constant or none.

`KeyCluster.suppress_markup`

Is true if key cluster suppresses key markup.

Returns boolean.

Special methods

`KeyCluster.__eq__(expr)`

Is true when *expr* is a key cluster indication with black-key and white-key inclusion equal to that of this key cluster indication. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`KeyCluster.__hash__()`

Hashes key cluster.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

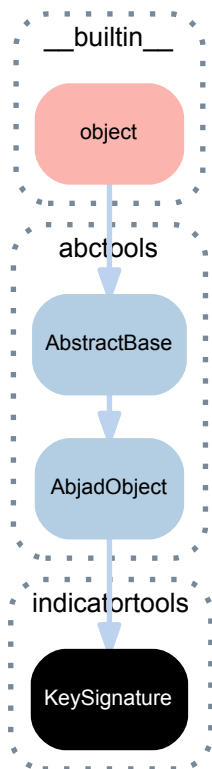
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.15 `indicatortools.KeySignature`



class `indicatortools.KeySignature` (*tonic='c', mode='major'*)
 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`KeySignature.mode`
 Mode of signature.

```

>>> key_signature.mode
Mode(mode_name='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__(expr)`

Is true when *expr* is a key signature with tonic and mode equal to that of this key signature. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`KeySignature.__hash__()`

Hashes key signature.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

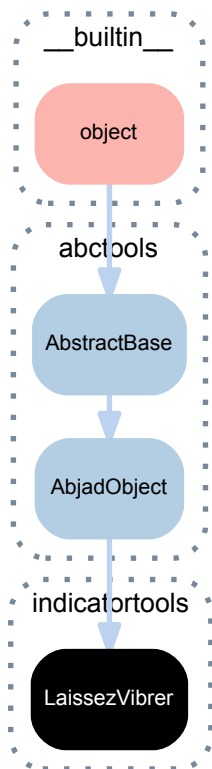
Returns string.

`KeySignature.__str__()`

String representation of key signature.

Returns string.

4.1.16 `indicatortools.LaissezVibrer`



class `indicatortools.LaissezVibrer`

A *laissez vibrer* indication.

```
>>> chord = Chord("<c' e' g' c''>4")
>>> laissez_vibrer = indicatortools.LaissezVibrer()
>>> attach(laissez_vibrer, chord)
>>> show(chord)
```



Bases

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

Special methods

`LaissezVibrer.__eq__(expr)`

Is true when *expr* is a *laissez vibrer* indication. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`LaissezVibrer.__hash__()`

Hashes laissez vibrer.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

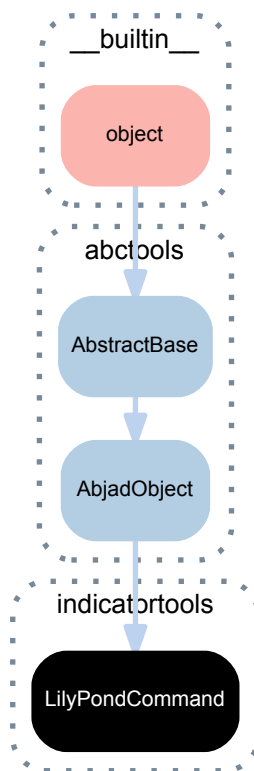
Returns string.

`LaissezVibrer.__str__()`

String representation of laissez vibrer.

Returns string.

4.1.17 `indicatortools.LilyPondCommand`



class `indicatortools.LilyPondCommand` (*name=None, format_slot=None*)

A LilyPond command.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.Slur()
>>> attach(slur, staff.select_leaves())
```

```
>>> command = indicatortools.LilyPondCommand('slurDotted')
>>> attach(command, staff[0])
```

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



Bases

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

Read-only properties

`LilyPondCommand.format_slot`

Gets format slot of LilyPond command.

```
>>> command.format_slot
'opening'
```

Returns string.

`LilyPondCommand.name`

Gets name of LilyPond command.

```
>>> command.name
'slurDotted'
```

Returns string.

Special methods

`LilyPondCommand.__copy__(*args)`

Copies LilyPond command.

Returns new LilyPond command.

`LilyPondCommand.__eq__(expr)`

Is true when *expr* 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.

`LilyPondCommand.__hash__()`

Hashes LilyPond command.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

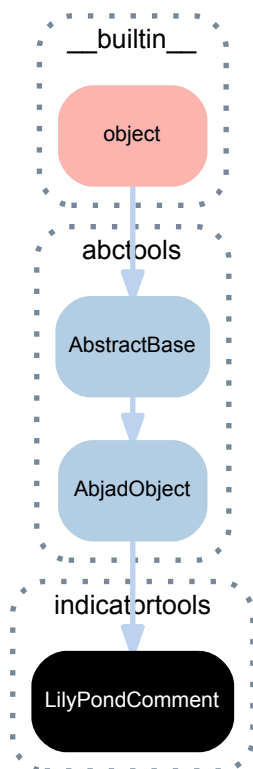
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.18 `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`
- `abctools.AbjadObject.AbstractBase`
- `__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__(expr)`

Is true when *expr* is a LilyPond comment with contents string equal to that of this LilyPond comment. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`LilyPondComment.__hash__()`

Hashes LilyPond comment.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

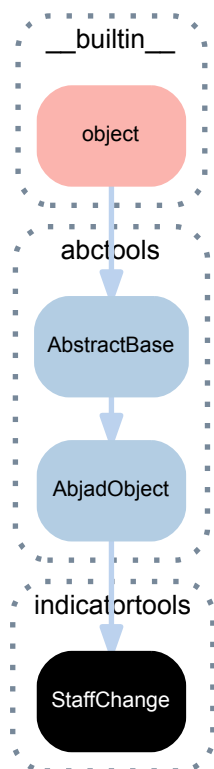
Returns string.

`LilyPondComment.__str__()`

Gets string format of LilyPond comment.

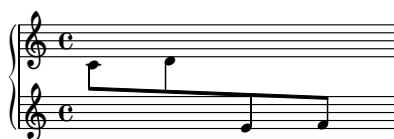
Returns string.

4.1.19 `indicatortools.StaffChange`



class `indicatortools.StaffChange` (*staff=None*)
 A staff change.

```
>>> staff_group = StaffGroup()
>>> staff_group.context_name = 'PianoStaff'
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> staff_group.extend([rh_staff, lh_staff])
>>> staff_change = indicatortools.StaffChange(lh_staff)
>>> attach(staff_change, rh_staff[2])
>>> show(staff_group)
```



Bases

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

Read-only properties

`StaffChange.staff`
 Staff of staff change.


```
>>> staff_change.staff
Staff('s2')
```

Returns staff.

Special methods

`StaffChange.__copy__(*args)`

Copies staff change.

Returns new staff change.

`StaffChange.__eq__(expr)`

Is true when *expr* is a staff change with a staff value equal to that of this staff change. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`StaffChange.__hash__()`

Hashes staff change.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

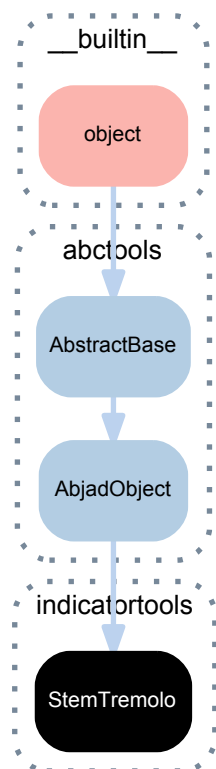
Returns string.

`StaffChange.__str__()`

Gets string format of staff change.

Returns string.

4.1.20 `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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`
Is true when *expr* is a stem tremolo with a tremolo flag count equal to that of this stem tremolo. 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.

`StemTremolo.__hash__()`
Hashes stem tremolo.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

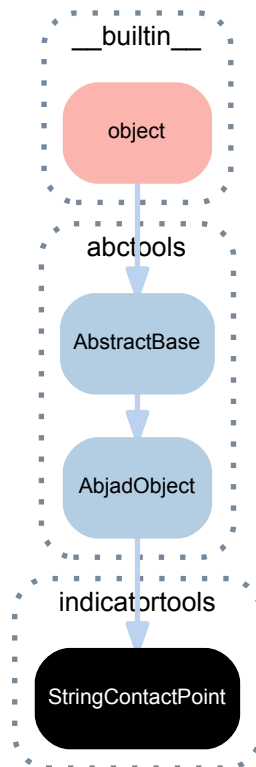
Returns boolean.

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

Returns string.

`StemTremolo.__str__()`
 String representation of stem tremolo.
 Returns string.

4.1.21 `indicatortools.StringContactPoint`



class `indicatortools.StringContactPoint` (*contact_point=None*)
 String contact point indicator.

```

>>> indicator = indicatortools.StringContactPoint('pont')
>>> print(format(indicator))
indicatortools.StringContactPoint(
    contact_point='pont',
)
    
```

Bases

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

Read-only properties

`StringContactPoint.contact_point`
 Gets contact point.

```

>>> indicator = indicatortools.StringContactPoint('tasto')
>>> indicator.contact_point
'tasto'
    
```

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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

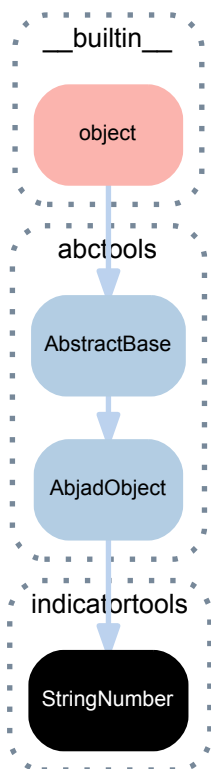
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.22 `indicatortools.StringNumber`



class `indicatortools.StringNumber` (*numbers=None*)

String number indicator.

```
>>> indicator = indicatortools.StringNumber(1)
>>> print(format(indicator))
indicatortools.StringNumber(
    numbers=(1,),
)

>>> indicator = indicatortools.StringNumber((2, 3))
>>> print(format(indicator))
indicatortools.StringNumber(
    numbers=(2, 3),
)
```

Bases

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

Read-only properties

`StringNumber.numbers`

Gets string numbers.

```
>>> indicator = indicatortools.StringNumber((2, 3))
>>> indicator.numbers
(2, 3)
```

`StringNumber.roman_numerals`

Gets string numbers as roman numerals.

```
>>> indicator = indicatortools.StringNumber((3, 4))
>>> indicator.roman_numerals
('iii', 'iv')
```

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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

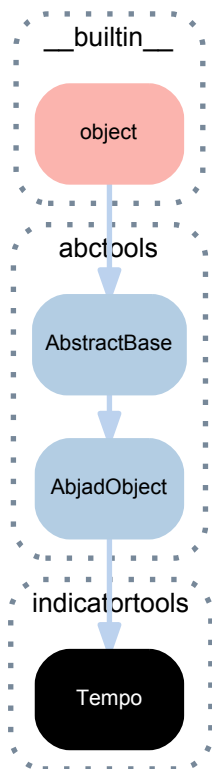
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

4.1.23 indicatortools.Tempo



class `indicatortools.Tempo` (*duration=None, units_per_minute=None, textual_indication=None*)
 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`
- `abctools.AbjadObject.AbstractBase`
- `__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(4, 60, 'Langsam').is_imprecise
False
>>> Tempo(textual_indication='Langsam').is_imprecise
True
>>> Tempo(4, (35, 50), 'Langsam').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.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)`
Is true when *expr* is a tempo with duration, textual indication and units-per-minute all equal to those of 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((1, 4), 84, 'Allegro')
>>> print(format(tempo))
indicatortools.Tempo(
    duration=durationtools.Duration(1, 4),
    units_per_minute=84,
    textual_indication='Allegro',
)
```

Returns string.

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

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

`Tempo.__hash__()`
Hashes tempo.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

`Tempo.__lt__(arg)`
Is 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=Duration(1, 4), units_per_minute=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=Duration(1, 4), units_per_minute=168)
```

Returns new tempo.

Tempo.**__str__**()

String representation of tempo.

```
>>> str(tempo)
'4=84'
```

Returns string.

Tempo.**__sub__**(*expr*)

Subtracts *expr* from tempo.

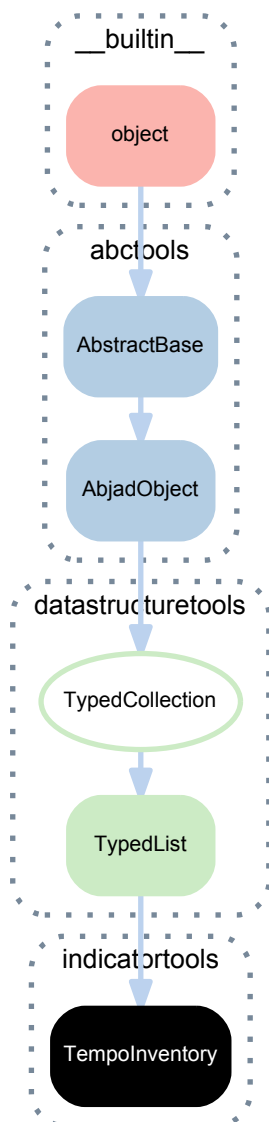
```
>>> tempo - 20
```

Returns new tempo.

Tempo.**__truediv__**(*expr*)

Divides tempo by *expr*. Operator for Python 3.

Returns new tempo.

4.1.24 `indicatortools.TempoInventory`

class `indicatortools.TempoInventory` (*items=None, item_class=None, keep_sorted=None*)
 An ordered list of tempo indications.

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

```
>>> for tempo in inventory:
...     tempo
...
Tempo(duration=Duration(1, 8), units_per_minute=72, textual_indication='Andante')
Tempo(duration=Duration(1, 8), units_per_minute=84, textual_indication='Allegro')
```

Tempo inventories implement list interface and are mutable.

Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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(item)`
Changes *item* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *item*)
Changes *item* 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** (*item*)
Changes *item* 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__** (*item*)
Is true when typed collection container *item*. Otherwise false.
Returns boolean.

(TypedList) **.__delitem__** (*i*)
Aliases list.__delitem__().
Returns none.

(TypedCollection) **.__eq__** (*expr*)
Is 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__**().

Returns item.

(TypedCollection).**__hash__**()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedList).**__iadd__**(*expr*)

Changes items 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(
            name='ppp',
        ),
        indicatortools.Dynamic(
            name='p',
        ),
        indicatortools.Dynamic(
            name='mp',
        ),
        indicatortools.Dynamic(
            name='mf',
        ),
        indicatortools.Dynamic(
            name='fff',
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

TempoInventory.**__illustrate__**()

Illustrates tempo inventory.

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

Andante (♩ = 72)

Allegro (♩ = 84)

Returns LilyPond file.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__**().

Returns generator.

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

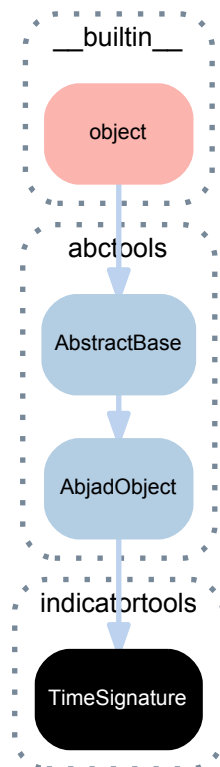
Changes items in *expr* to items and sets.

```
>>> 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,
)
```

Returns none.

4.1.25 `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`
- `abctools.AbjadObject.AbstractBase`
- `__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`
Is 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.implied_prolation`
Time signature implied prolation.

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

```
>>> TimeSignature((3, 8)).implied_prolation
Multiplier(1, 1)
```

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

```
>>> TimeSignature((7, 12)).implied_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.

Read/write properties

`TimeSignature.suppress`

Gets time signature suppression.

```
>>> time_signature.suppress is None
True
```

Sets time signature suppression.

```
>>> time_signature.suppress = True
```

Returns boolean or none.

Methods

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

Makes 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)`

Is 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)`

Is true when duration of time signature is greater than or equal to duration of *arg*. Otherwise false.

Returns boolean.

`TimeSignature.__gt__(arg)`

Is true when duration of time signature is greater than duration of *arg*. Otherwise false.

Returns boolean.

`TimeSignature.__hash__()`

Hashes time signature.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`TimeSignature.__le__(arg)`

Is true when duration of time signature is less than duration of *arg*. Otherwise false.

Returns boolean.

`TimeSignature.__lt__(arg)`

Is 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.

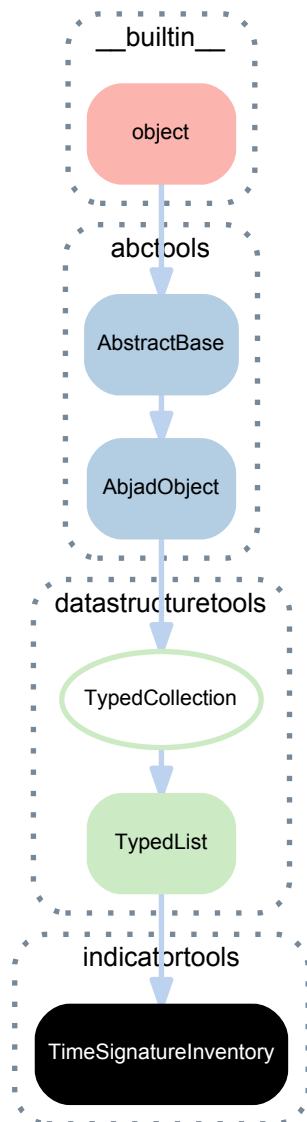
Returns string.

`TimeSignature.__str__()`

String representation of time signature.

Returns string.

4.1.26 `indicatortools.TimeSignatureInventory`



class `indicatortools.TimeSignatureInventory` (*items=None*, *item_class=None*,
keep_sorted=None)

An ordered list of time signatures.

```
>>> inventory = indicatortools.TimeSignatureInventory([(5, 8), (4, 4)])
```

```
>>> inventory
TimeSignatureInventory([TimeSignature((5, 8)), TimeSignature((4, 4))])
```

```
>>> (5, 8) in inventory
True
```

```
>>> TimeSignature((4, 4)) in inventory
True
```

```
>>> (3, 4) in inventory
False
```

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

5 ————— **10**

Time signature inventories implement the list interface and are mutable.

Bases

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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** (*item*)

Changes *item* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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__** (*item*)

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

Returns none.

(TypedCollection).**__eq__**(*expr*)

Is 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__**().

Returns item.

(TypedCollection).**__hash__**()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedList).**__iadd__**(*expr*)

Changes items 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(
            name='ppp',
        ),
        indicatortools.Dynamic(
            name='p',
        ),
        indicatortools.Dynamic(
            name='mp',
        ),
        indicatortools.Dynamic(
            name='mf',
        ),
        indicatortools.Dynamic(
            name='fff',
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

TimeSignatureInventory.**__illustrate__**(*format_specification*='')

Formats time signature inventory.

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

```
5-----|e-----|
```

Returns LilyPond file.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__**().

Returns generator.

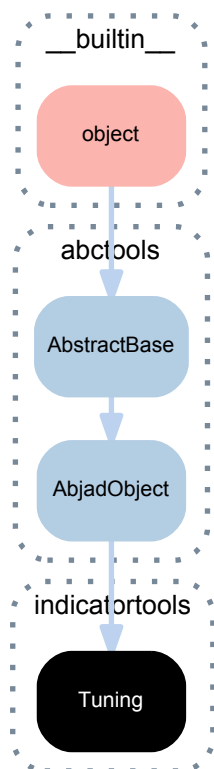
(TypedList).**__setitem__**(*i*, *expr*)

Changes items 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,
)
```

Returns none.

4.1.27 `indicatortools.Tuning`

class `indicatortools.Tuning` (*pitches=None*)
 Tuning indicator.

```

>>> indicator = indicatortools.Tuning(
...     pitches=('G3', 'D4', 'A4', 'E5'),
... )
>>> print(format(indicator))
indicatortools.Tuning(
    pitches=pitchtools.PitchSegment(
        (
            pitchtools.NamedPitch('g'),
            pitchtools.NamedPitch("d'"),
            pitchtools.NamedPitch("a'"),
            pitchtools.NamedPitch("e'"),
        ),
        item_class=pitchtools.NamedPitch,
    ),
)
  
```

Bases

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

Read-only properties**Tuning.pitch_ranges**

Gets two-octave pitch-ranges for each pitch in this tuning.

```

>>> pitch_ranges = indicator.pitch_ranges
>>> print(format(pitch_ranges))
pitchtools.PitchRangeInventory(
  
```

```
[
    pitchtools.PitchRange(
        range_string=' [G3, G5]',
    ),
    pitchtools.PitchRange(
        range_string=' [D4, D6]',
    ),
    pitchtools.PitchRange(
        range_string=' [A4, A6]',
    ),
    pitchtools.PitchRange(
        range_string=' [E5, E7]',
    ),
]
```

Returns pitch-range inventory.

Tuning.pitches

Gets tuning pitches.

```
>>> pitches = indicator.pitches
>>> print(format(pitches))
pitchtools.PitchSegment(
    (
        pitchtools.NamedPitch('g'),
        pitchtools.NamedPitch("d'"),
        pitchtools.NamedPitch("a'"),
        pitchtools.NamedPitch("e'"),
    ),
    item_class=pitchtools.NamedPitch,
)
```

Return pitch segment.

Methods

Tuning.get_pitch_ranges_by_string_number(string_number)

Gets tuning pitch ranges by string number.

```
>>> tuning = indicatortools.Tuning(('G3', 'D4', 'A4', 'E5'))
>>> string_number = indicatortools.StringNumber((2, 3))
>>> tuning.get_pitch_ranges_by_string_number(string_number)
(PitchRange(range_string=' [A4, A6]'), PitchRange(range_string=' [D4, D6]'))
```

Returns pitch ranges.

Tuning.get_pitches_by_string_number(string_number)

Gets tuning pitches by string number.

```
>>> tuning = indicatortools.Tuning(('G3', 'D4', 'A4', 'E5'))
>>> string_number = indicatortools.StringNumber((2, 3))
>>> tuning.get_pitches_by_string_number(string_number)
(NamedPitch("a'"), NamedPitch("d'"))
```

Returns named pitches.

Tuning.voice_pitch_classes(pitch_classes, allow_open_strings=True)

Voices *pitch_classes*.

```
>>> tuning = indicatortools.Tuning(('G3', 'D4', 'A4', 'E5'))
>>> voicings = tuning.voice_pitch_classes(('a',))
>>> for voicing in voicings:
...     voicing
...
(NamedPitch('a'), None, None, None)
(NamedPitch("a'"), None, None, None)
(None, NamedPitch("a'"), None, None)
(None, NamedPitch("a'"), None, None)
```


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

Formats Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

Returns boolean.

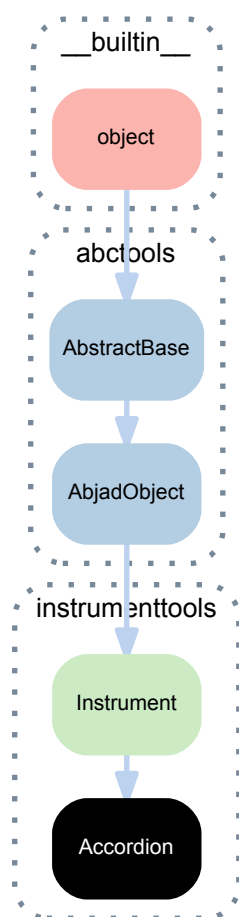
(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

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.

```
>>> staff_group = StaffGroup()
>>> staff_group.context_name = 'PianoStaff'
>>> staff_group.append(Staff("c'4 d'4 e'4 f'4"))
>>> staff_group.append(Staff("c'2 b2"))
```

```
>>> accordion = instrumenttools.Accordion()
>>> attach(accordion, staff_group)
>>> attach(Clef(name='bass'), staff_group[1])
>>> show(staff_group)
```



The accordion targets the piano staff context by default.

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

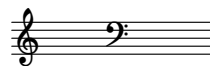
Read-only properties

`Accordion.allowable_clefs`

Gets accordion's allowable clefs.

```
>>> accordion.allowable_clefs
ClefInventory([Clef(name='treble'), Clef(name='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(contents=('Accordion',))
```

```
>>> show(accordion.instrument_name_markup)
```

Accordion

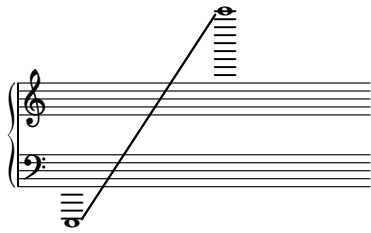
Returns markup.

`Accordion.pitch_range`

Gets accordion's range.

```
>>> accordion.pitch_range
PitchRange(range_string='[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(contents=('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(
    contents=('Accordion',),
),
short_instrument_name_markup=markuptools.Markup(
    contents=('Acc.',),
),
allowable_clefs=indicatortools.ClefInventory(
    [
        indicatortools.Clef(
            name='treble',
        ),
        indicatortools.Clef(
            name='bass',
        ),
    ]
),
pitch_range=pitchtools.PitchRange(
    range_string='[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.

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

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

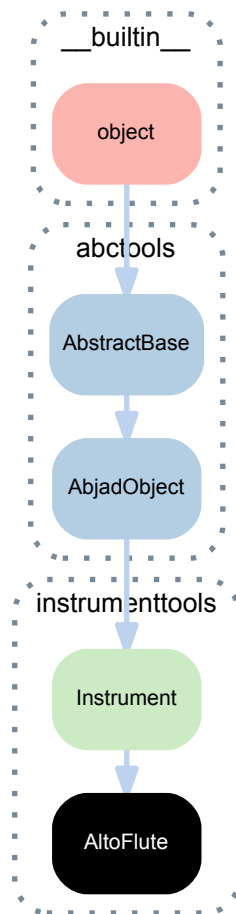
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.2 instrumenttools.AltoFlute

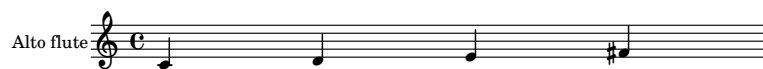


class `instrumenttools.AltoFlute` (*instrument_name='alto flute', short_instrument_name='alt. fl.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[G3, G6]', sounding_pitch_of_written_middle_c='G3'*)

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

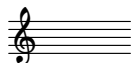
Read-only properties

AltoFlute.allowable_clefs

Gets alto flute's allowable clefs.

```
>>> alto_flute.allowable_clefs
ClefInventory([Clef(name='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(contents=('Alto flute',))
```

```
>>> show(alto_flute.instrument_name_markup)
```

Alto flute

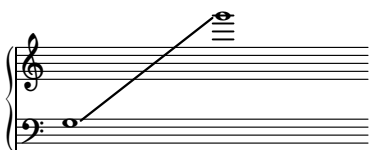
Returns markup.

AltoFlute.pitch_range

Gets alto flute's range.

```
>>> alto_flute.pitch_range
PitchRange(range_string='[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(contents=('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(
    contents=('Alto flute',),
  ),
  short_instrument_name_markup=markuptools.Markup(
    contents=('Alt. fl.',),
  ),
  allowable_clefs=indicatortools.ClefInventory(
    [
      indicatortools.Clef(
        name='treble',
      ),
    ]
  ),
  pitch_range=pitchtools.PitchRange(
    range_string='[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.

`(AbjadObject).__ne__(expr)`

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

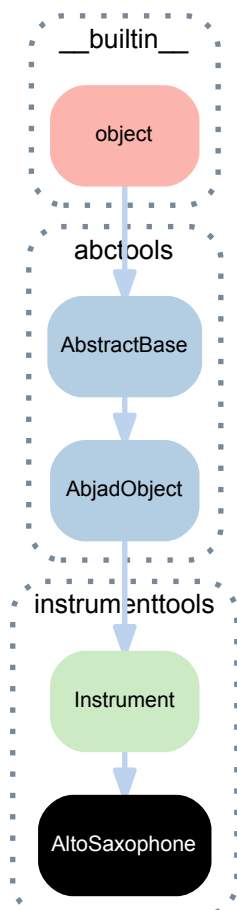
Returns boolean.

`(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, allow-
                                     able_clefs=None, pitch_range='[Db3, A5]', sound-
                                     ing_pitch_of_written_middle_c='Eb3')

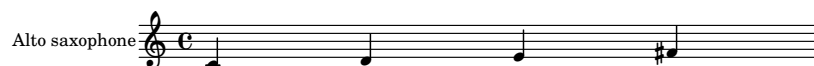
```

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`

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

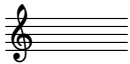
Read-only properties

`AltoSaxophone.allowable_clefs`

Gets alto saxophone's allowable clefs.

```
>>> alto_saxophone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Alto saxophone',))
```

```
>>> show(alto_saxophone.instrument_name_markup)
```

Alto saxophone

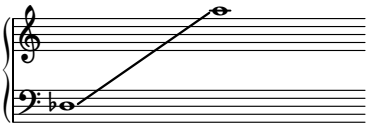
Returns markup.

`AltoSaxophone.pitch_range`

Gets alto saxophone's range.

```
>>> alto_saxophone.pitch_range
PitchRange(range_string='[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(contents=('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(
    contents=('Alto saxophone',),
  ),
  short_instrument_name_markup=markuptools.Markup(
    contents=('Alt. sax.',),
  ),
  allowable_clefs=indicatortools.ClefInventory(
    [
      indicatortools.Clef(
        name='treble',
      ),
    ]
  ),
  pitch_range=pitchtools.PitchRange(
    range_string='[Db3, A5]',
  ),
  sounding_pitch_of_written_middle_c=pitchtools.NamedPitch('ef'),
)
```

Returns string.

`(Instrument).__hash__()`

Gets hash value instrument.

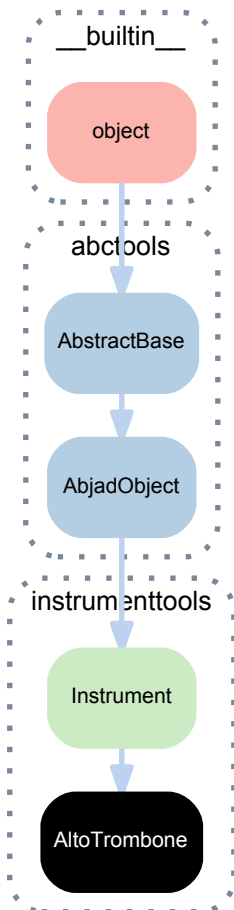
Computed on type, instrument name and short instrument name.

Returns integer.

(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(name='bass')
>>> attach(clef, staff)
>>> alto_trombone = instrumenttools.AltoTrombone()
>>> attach(alto_trombone, staff)
>>> show(staff)
```

Alto trombone

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`AltoTrombone.allowable_clefs`

Gets alto trombone's allowable clefs.

```
>>> alto_trombone.allowable_clefs
ClefInventory([Clef(name='bass'), Clef(name='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(contents=('Alto trombone',))
```

```
>>> show(alto_trombone.instrument_name_markup)
```

Alto trombone

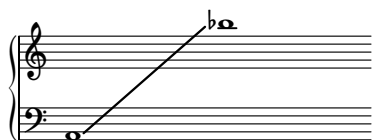
Returns markup.

`AltoTrombone.pitch_range`

Gets alto trombone's range.

```
>>> alto_trombone.pitch_range
PitchRange(range_string='[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(contents=('Alt. trb.',))
```

```
>>> show(alto_trombone.short_instrument_name_markup)
```

Alt. trb.

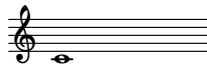
Returns markup.

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(
    contents=('Alto trombone',),
  ),
  short_instrument_name_markup=markuptools.Markup(
    contents=('Alt. trb.',),
  ),
  allowable_clefs=indicatortools.ClefInventory(
    [
      indicatortools.Clef(
        name='bass',
      ),
      indicatortools.Clef(
        name='tenor',
      ),
    ]
  ),
  pitch_range=pitchtools.PitchRange(
    range_string=' [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.

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

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

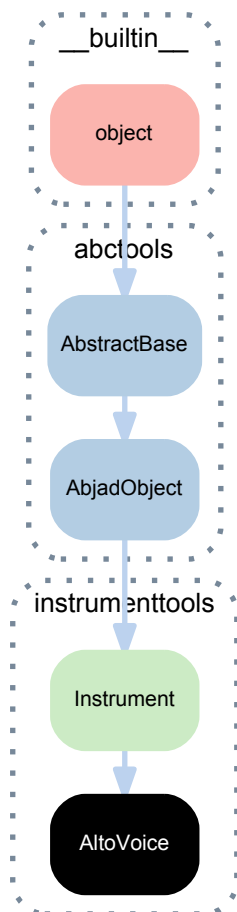
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.5 instrumenttools.AltoVoice



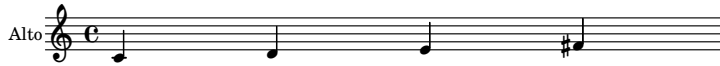
```

class instrumenttools.AltoVoice (instrument_name='alto',      short_instrument_name='alto',
                                instrument_name_markup=None,
                                short_instrument_name_markup=None,      allow-
                                able_clefs=None,      pitch_range='[F3,      G5]',      sound-
                                ing_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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

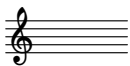
Read-only properties

`AltoVoice.allowable_clefs`

Gets alto's allowable clefs.

```
>>> alto.allowable_clefs
ClefInventory([Clef(name='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(contents=('Alto',))
```

```
>>> show(alto.instrument_name_markup)
```

Alto

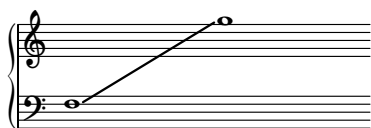
Returns markup.

`AltoVoice.pitch_range`

Gets alto's range.

```
>>> alto.pitch_range
PitchRange(range_string='[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(contents=('Alto',))
```

```
>>> show(alto.short_instrument_name_markup)
```

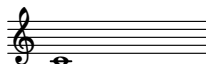
Alto

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.

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

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

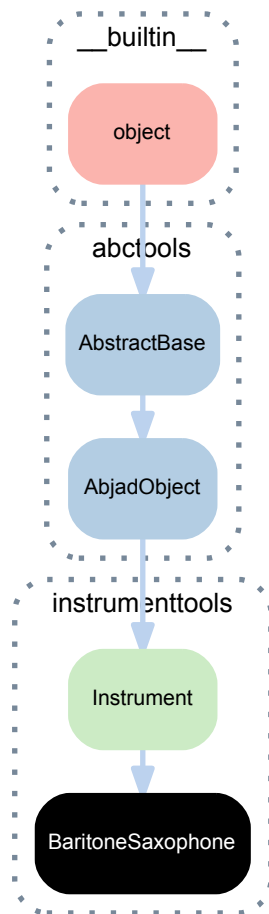
Returns boolean.

(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)

```

Baritone saxophone

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

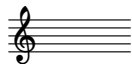
Read-only properties

BaritoneSaxophone.allowable_clefs

Gets baritone saxophone's allowable clefs.

```
>>> baritone_saxophone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Baritone saxophone',))
```

```
>>> show(baritone_saxophone.instrument_name_markup)
```

Baritone saxophone

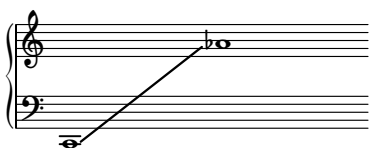
Returns markup.

BaritoneSaxophone.pitch_range

Gets baritone saxophone's range.

```
>>> baritone_saxophone.pitch_range
PitchRange(range_string='[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(contents=('Bar. sax.',))
```

```
>>> show(baritone_saxophone.short_instrument_name_markup)
```


Bar. sax.

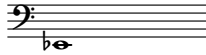
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

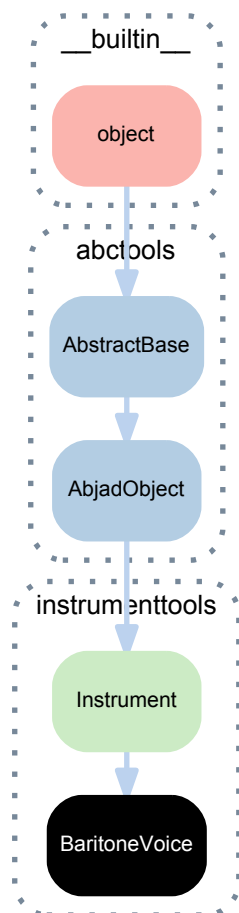
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.7 instrumenttools.BaritoneVoice



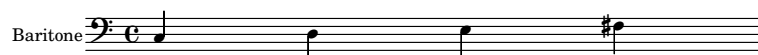
```

class instrumenttools.BaritoneVoice (instrument_name='baritone',
                                     short_instrument_name='bar.',
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,
                                     allowable_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(name='bass')
>>> attach(clef, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

BaritoneVoice.allowable_clefs

Gets baritone's allowable clefs.

```
>>> baritone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Baritone',))
```

```
>>> show(baritone.instrument_name_markup)
```

Baritone

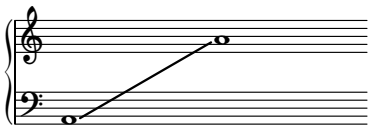
Returns markup.

BaritoneVoice.pitch_range

Gets baritone's range.

```
>>> baritone.pitch_range
PitchRange(range_string='[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(contents=('Bar.',))
```

```
>>> show(baritone.short_instrument_name_markup)
```

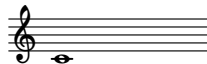
Bar.

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(baritone.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.

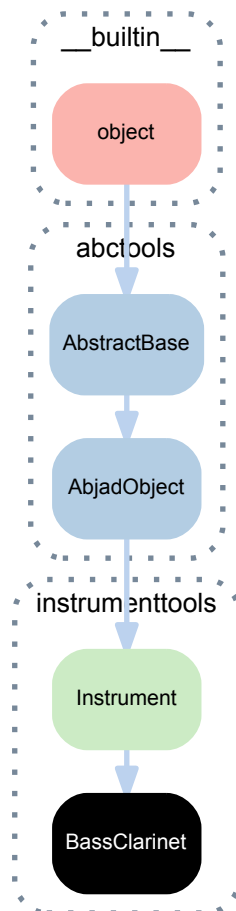
`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(Instrument).__repr__()`
 Gets interpreter representation of instrument.

Returns string.

5.1.8 instrumenttools.BassClarinet



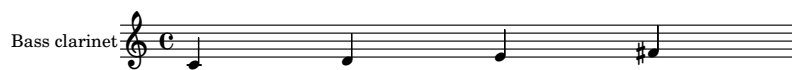
```

class instrumenttools.BassClarinet (instrument_name='bass',          clarinet',
                                   short_instrument_name='bass      cl.',
                                   instrument_name_markup=None,
                                   short_instrument_name_markup=None,
                                   allowable_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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

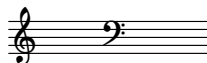
Read-only properties

BassClarinet.allowable_clefs

Gets bass clarinet's allowable clefs.

```
>>> bass_clarinet.allowable_clefs
ClefInventory([Clef(name='treble'), Clef(name='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(contents=('Bass clarinet',))
```

```
>>> show(bass_clarinet.instrument_name_markup)
```

Bass clarinet

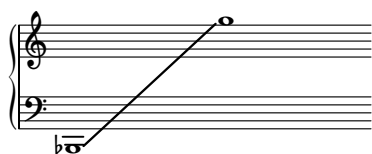
Returns markup.

BassClarinet.pitch_range

Gets bass clarinet's range.

```
>>> bass_clarinet.pitch_range
PitchRange(range_string='[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(contents=('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.

`(AbjadObject).__ne__(expr)`

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

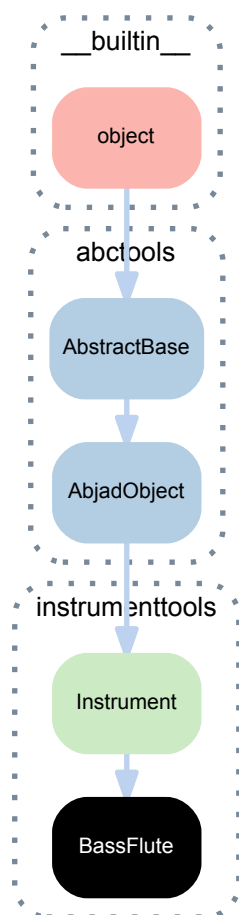
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.9 instrumenttools.BassFlute

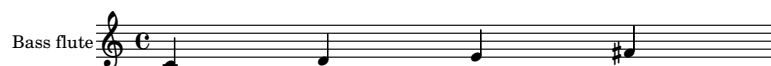


class `instrumenttools.BassFlute` (*instrument_name='bass flute', short_instrument_name='bass fl.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[C3, C6]', sounding_pitch_of_written_middle_c='C3'*)

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

Read-only properties

`BassFlute.allowable_clefs`

Gets bass flute's allowable clefs.

```
>>> bass_flute.allowable_clefs
ClefInventory([Clef(name='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(contents=('Bass flute',))
```

```
>>> show(bass_flute.instrument_name_markup)
```

Bass flute

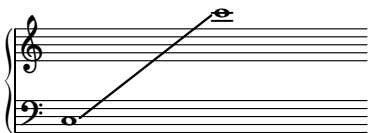
Returns markup.

`BassFlute.pitch_range`

Gets bass flute's range.

```
>>> bass_flute.pitch_range
PitchRange(range_string='[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(contents=('Bass fl.',))
```

```
>>> show(bass_flute.short_instrument_name_markup)
```

Bass fl.

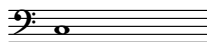
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

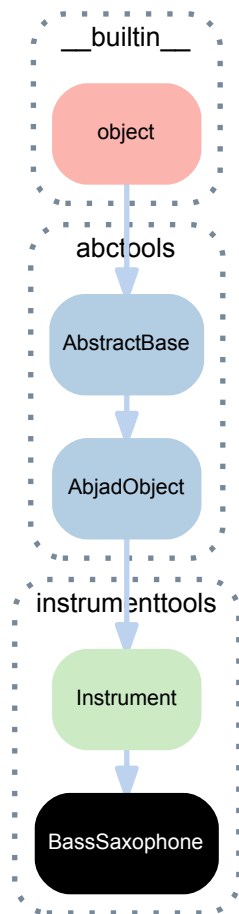
Returns boolean.

`(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)
```

Bass saxophone

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

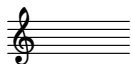
Read-only properties

`BassSaxophone.allowable_clefs`

Gets bass saxophone's allowable clefs.

```
>>> bass_saxophone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Bass saxophone',))
```

```
>>> show(bass_saxophone.instrument_name_markup)
```

Bass saxophone

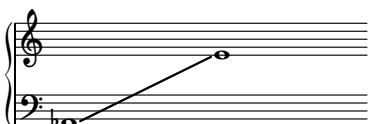
Returns markup.

`BassSaxophone.pitch_range`

Gets bass saxophone's range.

```
>>> bass_saxophone.pitch_range
PitchRange(range_string='[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(contents=('Bass sax.',))
```

```
>>> show(bass_saxophone.short_instrument_name_markup)
```

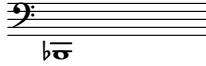
Bass sax.

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.

`(AbjadObject).__ne__(expr)`

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

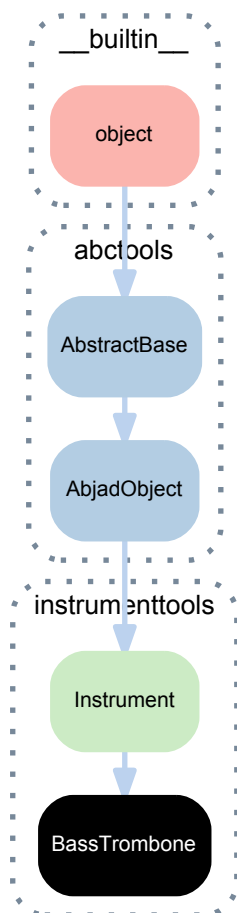
Returns boolean.

`(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(name='bass')
>>> attach(clef, staff)
>>> bass_trombone = instrumenttools.BassTrombone()
>>> attach(bass_trombone, staff)
>>> show(staff)

```

Bass trombone

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

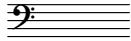
Read-only properties

BassTrombone.allowable_clefs

Gets bass trombone's allowable clefs.

```
>>> bass_trombone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Bass trombone',))
```

```
>>> show(bass_trombone.instrument_name_markup)
```

Bass trombone

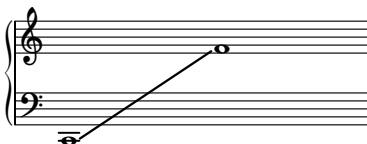
Returns markup.

BassTrombone.pitch_range

Gets bass trombone's range.

```
>>> bass_trombone.pitch_range
PitchRange(range_string='[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(contents=('Bass trb.',))
```

```
>>> show(bass_trombone.short_instrument_name_markup)
```

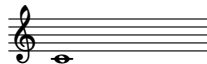
Bass trb.

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.

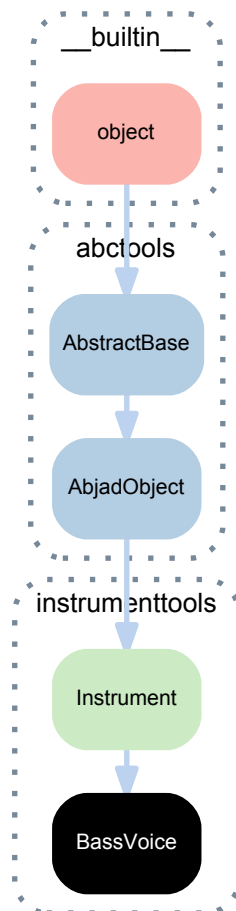
`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(Instrument).__repr__()`
 Gets interpreter representation of instrument.

Returns string.

5.1.12 instrumenttools.BassVoice

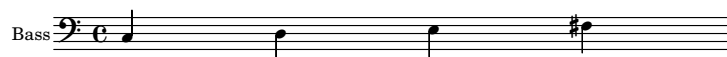


class `instrumenttools.BassVoice` (*instrument_name='bass', short_instrument_name='bass', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('bass',), pitch_range='[E2, F4]', sounding_pitch_of_written_middle_c=None*)

A bass.

```

>>> staff = Staff("c4 d4 e4 fs4")
>>> bass = instrumenttools.BassVoice()
>>> attach(bass, staff)
>>> clef = Clef(name='bass')
>>> attach(clef, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

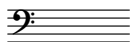
Read-only properties

BassVoice.allowable_clefs

Gets bass's allowable clefs.

```
>>> bass.allowable_clefs
ClefInventory([Clef(name='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(contents=('Bass',))
```

```
>>> show(bass.instrument_name_markup)
```

Bass

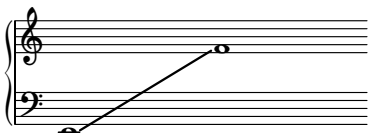
Returns markup.

BassVoice.pitch_range

Gets bass's range.

```
>>> bass.pitch_range
PitchRange(range_string='[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(contents=('Bass',))
```

```
>>> show(bass.short_instrument_name_markup)
```

Bass

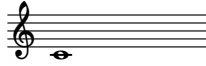
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

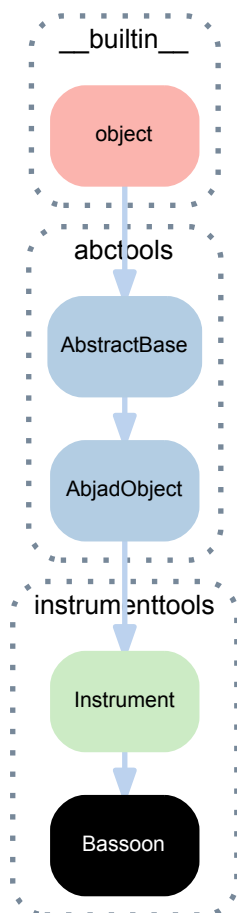
Returns boolean.

`(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,
                             allowable_clefs=('bass', 'tenor'), pitch_range='[Bb1, Eb5]',
                             sounding_pitch_of_written_middle_c=None)

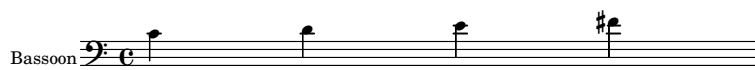
```

A bassoon.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef(name='bass')
>>> attach(clef, staff)
>>> bassoon = instrumenttools.Bassoon()
>>> attach(bassoon, staff)
>>> show(staff)

```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

Bassoon.allowable_clefs

Gets bassoon's allowable clefs.

```
>>> bassoon.allowable_clefs
ClefInventory([Clef(name='bass'), Clef(name='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(contents=('Bassoon',))
```

```
>>> show(bassoon.instrument_name_markup)
```

Bassoon

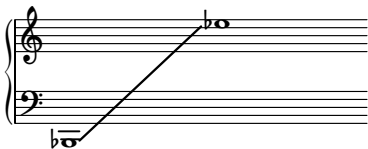
Returns markup.

Bassoon.pitch_range

Gets bassoon's range.

```
>>> bassoon.pitch_range
PitchRange(range_string='[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(contents=('Bsn.',))
```

```
>>> show(bassoon.short_instrument_name_markup)
```

Bsn.

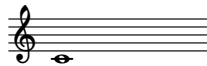
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

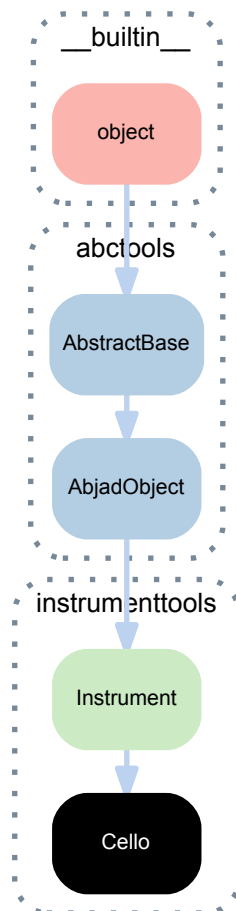
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.14 instrumenttools.Cello

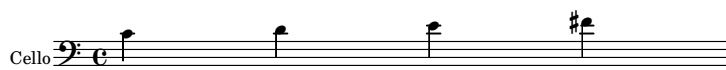


class `instrumenttools.Cello` (*instrument_name='cello', short_instrument_name='vc.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('bass', 'tenor', 'treble'), default_tuning=('C2', 'G2', 'D3', 'A3'), pitch_range='[C2, G5]', sounding_pitch_of_written_middle_c=None*)

A cello.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef(name='bass')
>>> attach(clef, staff)
>>> cello = instrumenttools.Cello()
>>> attach(cello, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

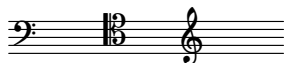
Read-only properties

Cello.allowable_clefs

Gets cello's allowable clefs.

```
>>> cello.allowable_clefs
ClefInventory([Clef(name='bass'), Clef(name='tenor'), Clef(name='treble')])
```

```
>>> show(cello.allowable_clefs)
```



Returns clef inventory.

Cello.default_tuning

Gets cello's default tuning.

```
>>> cello.default_tuning
Tuning(pitches=PitchSegment(['c,', 'g,', 'd', 'a']))
```

Returns tuning.

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(contents=('Cello',))
```

```
>>> show(cello.instrument_name_markup)
```

Cello

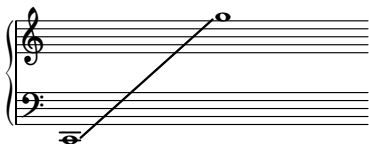
Returns markup.

Cello.pitch_range

Gets cello's range.

```
>>> cello.pitch_range
PitchRange(range_string='[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(contents=('Vc.',))
```

```
>>> show(cello.short_instrument_name_markup)
```

Vc.

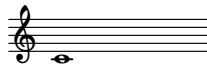
Returns markup.

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.

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

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

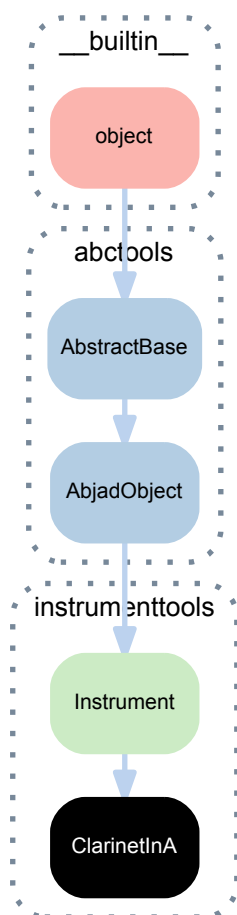
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

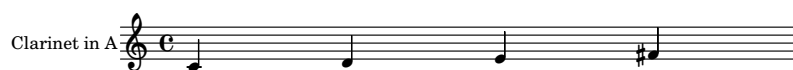
5.1.15 instrumenttools.ClarinetInA



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

Read-only properties

ClarineteInA.allowable_clefs

Gets clarinet in A's allowable clefs.

```
>>> clarinet.allowable_clefs
ClefInventory([Clef(name='treble')])
```

```
>>> show(clarinet.allowable_clefs)
```



Returns clef inventory.

ClarineteInA.instrument_name

Gets clarinet in A's name.

```
>>> clarinet.instrument_name
'clarinet in A'
```

Returns string.

ClarineteInA.instrument_name_markup

Gets clarinet in A's instrument name markup.

```
>>> clarinet.instrument_name_markup
Markup(contents=('Clarinet in A',))
```

```
>>> show(clarinet.instrument_name_markup)
```

Clarinet in A

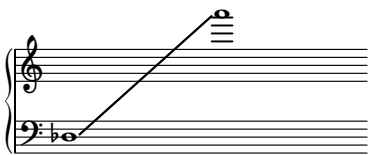
Returns markup.

ClarineteInA.pitch_range

Gets clarinet in A's range.

```
>>> clarinet.pitch_range
PitchRange(range_string='[Db3, A6]')
```

```
>>> show(clarinet.pitch_range)
```



Returns pitch range.

ClarineteInA.short_instrument_name

Gets clarinet in A's short instrument name.

```
>>> clarinet.short_instrument_name
'cl. A \\natural'
```

Returns string.

ClarineteInA.short_instrument_name_markup

Gets clarinet in A's short instrument name markup.

```
>>> clarinet.short_instrument_name_markup
Markup(contents=('Cl.', 'A', MarkupCommand('natural')))
```

```
>>> show(clarinet.short_instrument_name_markup)
```

Cl. A 

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.

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

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

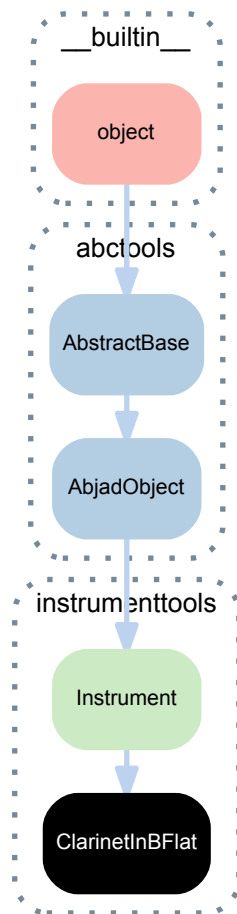
Returns boolean.

(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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

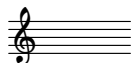
Read-only properties

`ClarinetInBFlat.allowable_clefs`

Gets clarinet in B-flat's allowable clefs.

```
>>> clarinet.allowable_clefs
ClefInventory([Clef(name='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(contents=('Clarinet in B-flat',))
```

```
>>> show(clarinet.instrument_name_markup)
```

Clarinet in B-flat

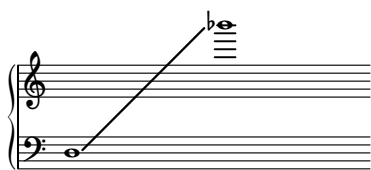
Returns markup.

`ClarinetInBFlat.pitch_range`

Gets clarinet in B-flat's range.

```
>>> clarinet.pitch_range
PitchRange(range_string='[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(contents=('Cl. in B-flat',))
```

```
>>> show(clarinet.short_instrument_name_markup)
```

Cl. in B-flat

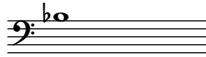
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

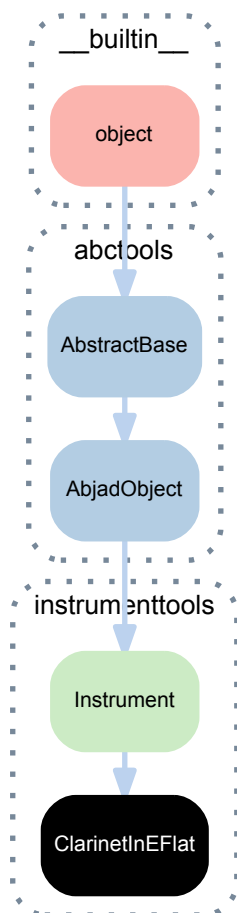
Returns boolean.

`(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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

ClarinetInEFlat.allowable_clefs

Gets clarinet in E-flat's allowable clefs.

```
>>> clarinet.allowable_clefs
ClefInventory([Clef(name='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(contents=('Clarinet in E-flat',))
```

```
>>> show(clarinet.instrument_name_markup)
```

Clarinet in E-flat

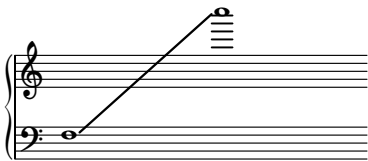
Returns markup.

ClarinetInEFlat.pitch_range

Gets clarinet in E-flat's range.

```
>>> clarinet.pitch_range
PitchRange(range_string='[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(contents=('Cl. E-flat',))
```

```
>>> show(clarinet.short_instrument_name_markup)
```

Cl. E-flat

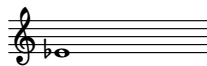
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

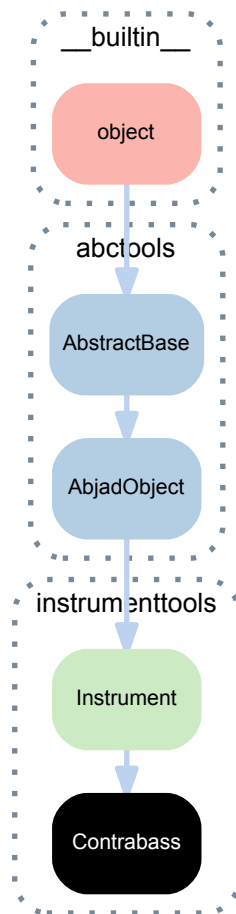
Returns boolean.

`(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'),
                                  default_tuning=('C1', 'A1', 'D2', 'G2'),
                                  pitch_range='[C1, G4]',
                                  sounding_pitch_of_written_middle_c='C3')
  
```

A contrabass.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef(name='bass')
>>> attach(clef, staff)
>>> contrabass = instrumenttools.Contrabass()
>>> attach(contrabass, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

Contrabass.allowable_clefs

Gets contrabass's allowable clefs.

```
>>> contrabass.allowable_clefs
ClefInventory([Clef(name='bass'), Clef(name='treble')])
```

```
>>> show(contrabass.allowable_clefs)
```



Returns clef inventory.

Contrabass.default_tuning

Gets contrabass's default tuning.

```
>>> contrabass.default_tuning
Tuning(pitches=PitchSegment(['c,,', 'a,,', 'd,', 'g,']))
```

Returns tuning.

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(contents=('Contrabass',))
```

```
>>> show(contrabass.instrument_name_markup)
```

Contrabass

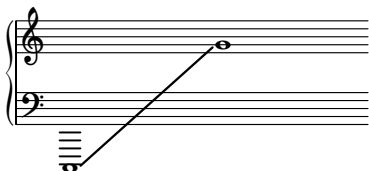
Returns markup.

Contrabass.pitch_range

Gets contrabass's range.

```
>>> contrabass.pitch_range
PitchRange(range_string='[C1, G4]')
```

```
>>> 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(contents=('Vb.',))
```

```
>>> show(contrabass.short_instrument_name_markup)
```

Vb.

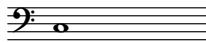
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

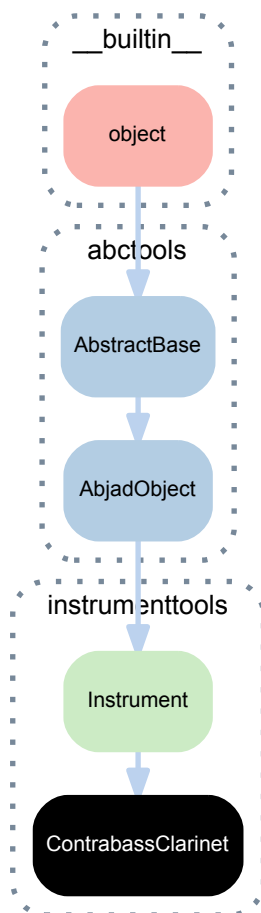
Returns boolean.

`(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)
  
```

Contrabass clarinet

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`ContrabassClarinet.allowable_clefs`

Gets contrabass clarinet's allowable clefs.

```
>>> contrabass_clarinet.allowable_clefs
ClefInventory([Clef(name='treble'), Clef(name='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(contents=('Contrabass clarinet',))
```

```
>>> show(contrabass_clarinet.instrument_name_markup)
```

Contrabass clarinet

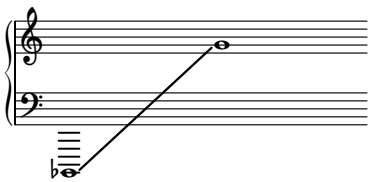
Returns markup.

`ContrabassClarinet.pitch_range`

Gets contrabass clarinet's range.

```
>>> contrabass_clarinet.pitch_range
PitchRange(range_string='[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(contents=('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.

`(AbjadObject).__ne__(expr)`

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

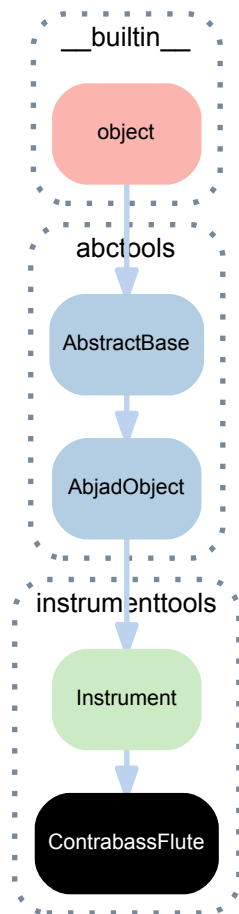
Returns boolean.

`(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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

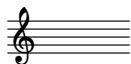
Read-only properties

`ContrabassFlute.allowable_clefs`

Gets contrabass flute's allowable clefs.

```
>>> contrabass_flute.allowable_clefs
ClefInventory([Clef(name='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(contents=('Contrabass flute',))
```

```
>>> show(contrabass_flute.instrument_name_markup)
```

Contrabass flute

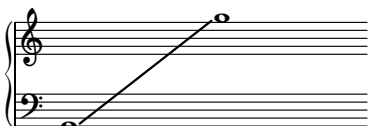
Returns markup.

`ContrabassFlute.pitch_range`

Gets contrabass flute's range.

```
>>> contrabass_flute.pitch_range
PitchRange(range_string='[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(contents=('Cbass. fl.',))
```

```
>>> show(contrabass_flute.short_instrument_name_markup)
```

Cbass. fl.

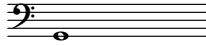
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

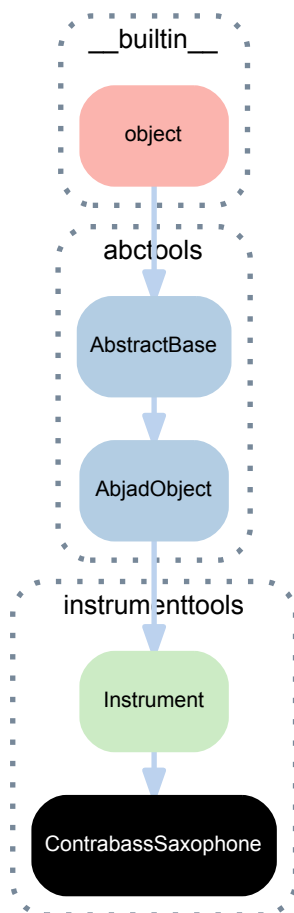
Returns boolean.

`(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)
  
```

Contrabass saxophone

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

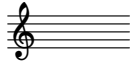
Read-only properties

ContrabassSaxophone.allowable_clefs

Gets contrabass saxophone's allowable clefs.

```
>>> contrabass_saxophone.allowable_clefs
ClefInventory([Clef(name='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(contents=('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(range_string='[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(contents=('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.

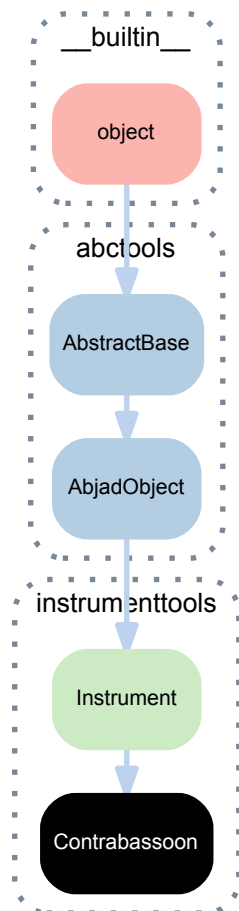
`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.

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,
                                     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(name='bass')
>>> attach(clef, staff)
>>> contrabassoon = instrumenttools.Contrabassoon()
>>> attach(contrabassoon, staff)
>>> show(staff)
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`Contrabassoon.allowable_clefs`

Gets contrabassoon's allowable clefs.

```
>>> contrabassoon.allowable_clefs
ClefInventory([Clef(name='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(contents=('Contrabassoon',))
```

```
>>> show(contrabassoon.instrument_name_markup)
```

Contrabassoon

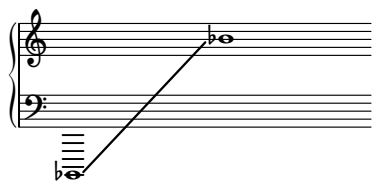
Returns markup.

`Contrabassoon.pitch_range`

Gets contrabassoon's range.

```
>>> contrabassoon.pitch_range
PitchRange(range_string='[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(contents=('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.

`(AbjadObject).__ne__(expr)`

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

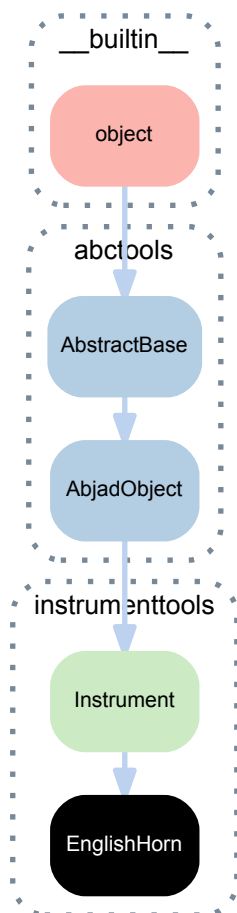
Returns boolean.

`(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,
                                   able_clefs=None, pitch_range='[E3, C6]', sound-
                                   ing_pitch_of_written_middle_c='F3')

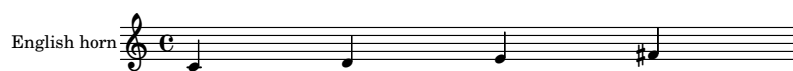
```

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

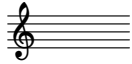
Read-only properties

EnglishHorn.allowable_clefs

Gets English horn's allowable clefs.

```
>>> english_horn.allowable_clefs
ClefInventory([Clef(name='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(contents=('English horn',))
```

```
>>> show(english_horn.instrument_name_markup)
```

English horn

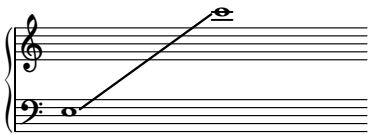
Returns markup.

EnglishHorn.pitch_range

Gets English horn's range.

```
>>> english_horn.pitch_range
PitchRange(range_string='[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(contents=('Eng. hn.',))
```

```
>>> show(english_horn.short_instrument_name_markup)
```

Eng. hn.

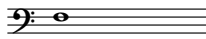
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

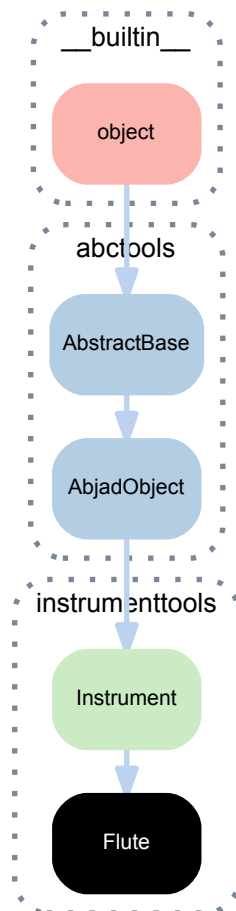
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.24 instrumenttools.Flute

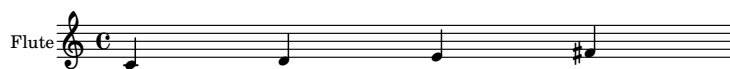


class instrumenttools.**Flute** (*instrument_name='flute', short_instrument_name='fl.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[C4, D7]', sounding_pitch_of_written_middle_c=None*)

A flute.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> flute = instrumenttools.Flute()
>>> attach(flute, staff)
>>> show(staff)
  
```



Bases

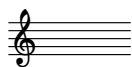
- instrumenttools.Instrument
- abctools.AbjadObject
- abctools.AbjadObject.AbstractBase
- __builtin__.object

Read-only properties

Flute.allowable_clefs
Gets flute's allowable clefs.

```
>>> flute.allowable_clefs
ClefInventory ([Clef (name='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 (contents=('Flute',))
```

```
>>> show(flute.instrument_name_markup)
```

Flute

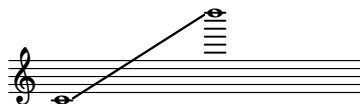
Returns markup.

Flute.pitch_range

Gets flute's range.

```
>>> flute.pitch_range
PitchRange (range_string=' [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 (contents=('Fl.',))
```

```
>>> show(flute.short_instrument_name_markup)
```

Fl.

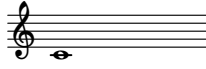
Returns markup.

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.

```
(AbjadObject).__ne__(expr)
```

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

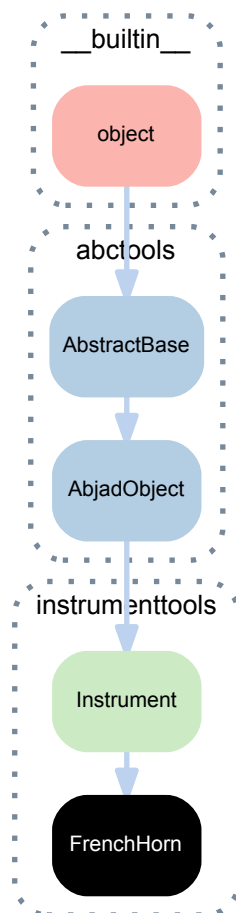
Returns boolean.

```
(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,    allow-
                                able_clefs=('bass', 'treble'), pitch_range='[B1, F5]',
                                sounding_pitch_of_written_middle_c='F3')

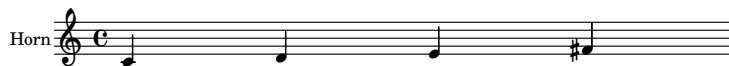
```

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

Read-only properties

FrenchHorn.allowable_clefs

Gets French horn's allowable clefs.

```
>>> french_horn.allowable_clefs
ClefInventory([Clef(name='bass'), Clef(name='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(contents=('Horn',))
```

```
>>> show(french_horn.instrument_name_markup)
```

Horn

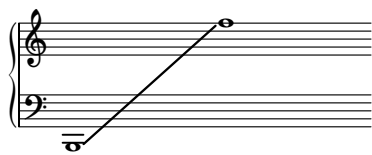
Returns markup.

FrenchHorn.pitch_range

Gets French horn's range.

```
>>> french_horn.pitch_range
PitchRange(range_string='[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(contents=('Hn.',))
```

```
>>> show(french_horn.short_instrument_name_markup)
```

Hn.

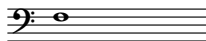
Returns markup.

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.

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

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

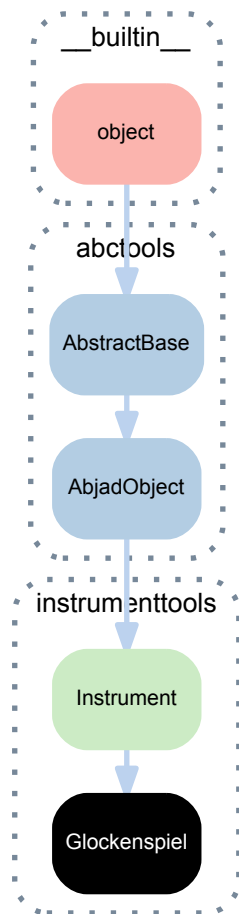
Returns boolean.

(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-
                                     strument_name_markup=None,
                                     short_instrument_name_markup=None,       allow-
                                     able_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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

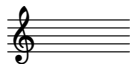
Read-only properties

Glockenspiel.allowable_clefs

Gets glockenspiel's allowable clefs.

```
>>> glockenspiel.allowable_clefs
ClefInventory([Clef(name='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(contents=('Glockenspiel',))
```

```
>>> show(glockenspiel.instrument_name_markup)
```

Glockenspiel

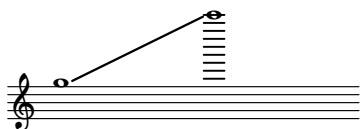
Returns markup.

Glockenspiel.pitch_range

Gets glockenspiel's range.

```
>>> glockenspiel.pitch_range
PitchRange(range_string='[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(contents=('Gkspl.',))
```

```
>>> show(glockenspiel.short_instrument_name_markup)
```

Gkspl.

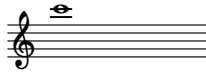
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

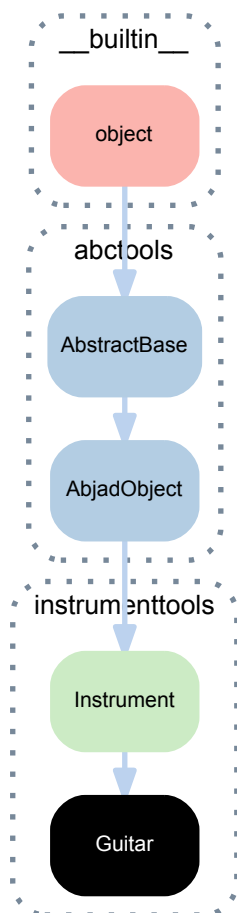
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.27 instrumenttools.Guitar



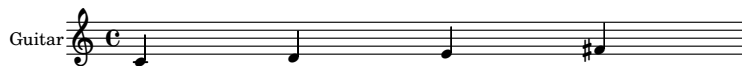
class instrumenttools.**Guitar** (*instrument_name='guitar', short_instrument_name='gt.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, default_tuning=('E2', 'A2', 'D3', 'G3', 'B3', 'E4'), pitch_range='[E2, E5]', sounding_pitch_of_written_middle_c='C3'*)

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
- abctools.AbjadObject.AbstractBase
- __builtin__.object

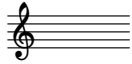
Read-only properties

Guitar.**allowable_clefs**

Gets guitar's allowable clefs.

```
>>> guitar.allowable_clefs
ClefInventory([Clef(name='treble')])
```

```
>>> show(guitar.allowable_clefs)
```



Returns clef inventory.

Guitar.**default_tuning**

Gets guitar's default tuning.

```
>>> guitar.default_tuning
Tuning(pitches=PitchSegment(['e','a','d','g','b','e']))
```

Returns tuning.

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(contents=('Guitar',))
```

```
>>> show(guitar.instrument_name_markup)
```

Guitar

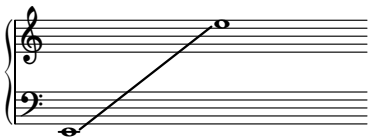
Returns markup.

Guitar.**pitch_range**

Gets guitar's range.

```
>>> guitar.pitch_range
PitchRange(range_string='[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(contents=('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.

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

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

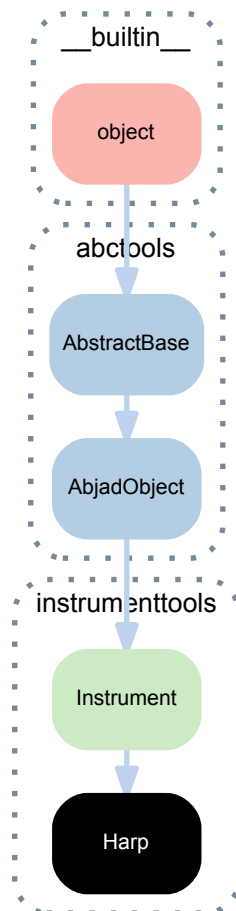
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

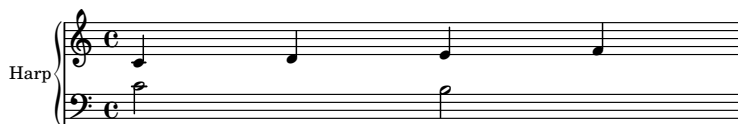
5.1.28 instrumenttools.Harp



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.

```
>>> staff_group = StaffGroup()
>>> staff_group.context_name = 'PianoStaff'
>>> staff_group.append(Staff("c'4 d'4 e'4 f'4"))
>>> staff_group.append(Staff("c'2 b2"))
>>> harp = instrumenttools.Harp()
>>> attach(harp, staff_group)
>>> attach(Clef(name='bass'), staff_group[1])
>>> show(staff_group)
```



The harp targets piano staff context by default.

Bases

- instrumenttools.Instrument
- abctools.AbjadObject
- abctools.AbjadObject.AbstractBase

- `__builtin__.object`

Read-only properties

`Harp.allowable_clefs`

Gets harp's allowable clefs.

```
>>> harp.allowable_clefs
ClefInventory([Clef(name='treble'), Clef(name='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(contents=('Harp',))
```

```
>>> show(harp.instrument_name_markup)
```

Harp

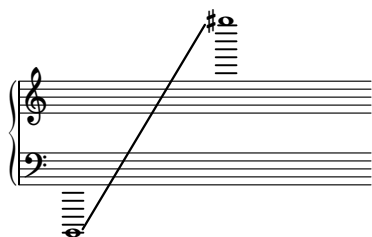
Returns markup.

`Harp.pitch_range`

Gets harp's range.

```
>>> harp.pitch_range
PitchRange(range_string='[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(contents=('Hp.',))
```

```
>>> show(harp.short_instrument_name_markup)
```

Hp.

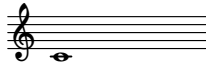
Returns markup.

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.

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

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

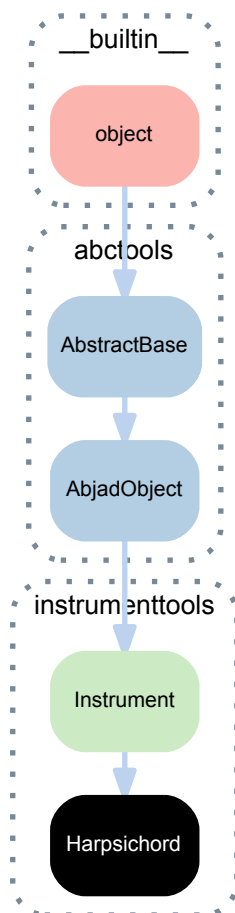
Returns boolean.

(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,
                                  allowable_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")
>>> staff_group = StaffGroup([upper_staff, lower_staff])
>>> staff_group.context_name = 'PianoStaff'
>>> harpsichord = instrumenttools.Harpsichord()
>>> attach(harpsichord, staff_group)
>>> attach(Clef(name='bass'), lower_staff)
>>> show(staff_group)
```



The harpsichord targets piano staff context by default.

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`

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

Read-only properties

`Harpsichord.allowable_clefs`

Gets harpsichord's allowable clefs.

```
>>> harpsichord.allowable_clefs
ClefInventory([Clef(name='treble'), Clef(name='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(contents=(' Harpsichord',))
```

```
>>> show(harpsichord.instrument_name_markup)
```

Harpsichord

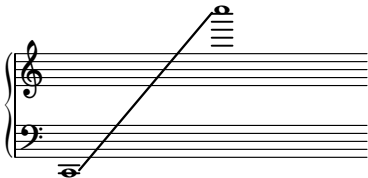
Returns markup.

`Harpsichord.pitch_range`

Gets harpsichord's range.

```
>>> harpsichord.pitch_range
PitchRange(range_string='[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(contents=('Hpschd.',))
```

```
>>> show(harpsichord.short_instrument_name_markup)
```

Hpschd.

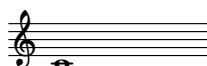
Returns markup.

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.

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

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

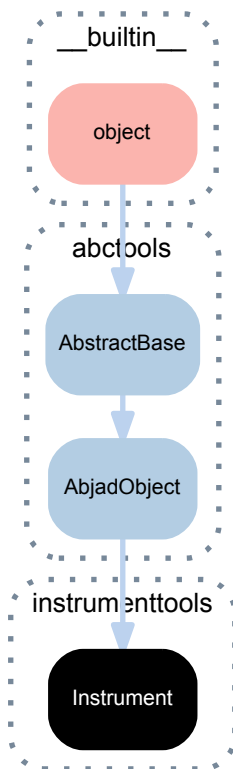
Returns boolean.

(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, allowable_clefs=None, pitch_range=None, allowing_pitch_of_written_middle_c=None*)

A musical instrument.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

`(AbjadObject).__ne__(expr)`

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

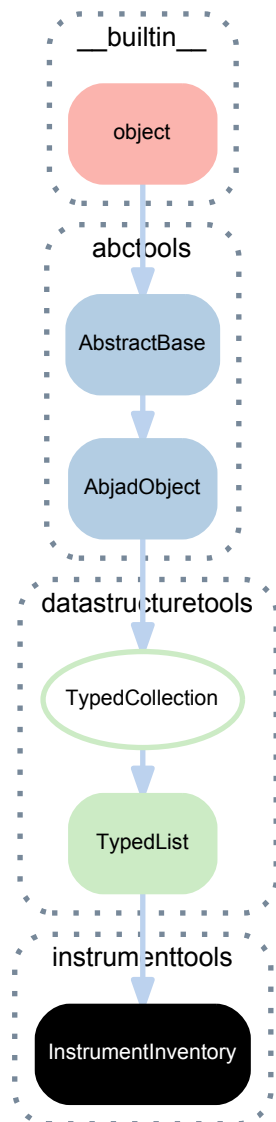
Returns boolean.

`Instrument.__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.31 instrumenttools.InstrumentInventory



class `instrumenttools.InstrumentInventory` (*items=None*, *item_class=None*, *keep_sorted=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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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(item)`
Changes *item* to item and returns index.

```
>>> pitch_collection = datastructuretools.TypedList(  
...     items=('cqf', "as'", 'b', 'dss'),  
...     item_class=NamedPitch)  
>>> pitch_collection.index("as'")  
1
```

Returns index.

`(TypedList) .insert (i, item)`
 Changes *item* 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 (item)`
 Changes *item* 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__ (item)`
 Is true when typed collection container *item*. Otherwise false.

Returns boolean.

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

Returns none.

`(TypedCollection) .__eq__ (expr)`
 Is 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__**().

Returns item.

(TypedCollection).**__hash__**()
 Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedList).**__iadd__**(*expr*)
 Changes items 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(
            name='ppp',
        ),
        indicatortools.Dynamic(
            name='p',
        ),
        indicatortools.Dynamic(
            name='mp',
        ),
        indicatortools.Dynamic(
            name='mf',
        ),
        indicatortools.Dynamic(
            name='fff',
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

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

Returns generator.

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

Returns nonnegative integer.

(TypedCollection).**__ne__**(*expr*)
 Is 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__**().

Returns generator.

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

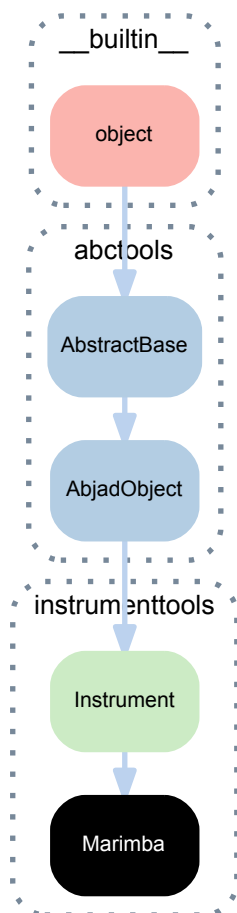
Changes items in *expr* to items and sets.

```
>>> 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,
)
```

Returns none.

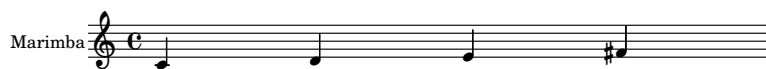
5.1.32 instrumenttools.Marimba



```
class instrumenttools.Marimba(instrument_name='marimba',    short_instrument_name='mb.',
                                instrument_name_markup=None,
                                short_instrument_name_markup=None,          allow-
                                able_clefs=('treble', 'bass'), pitch_range='[F2, C7]', sound-
                                ing_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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`Marimba.allowable_clefs`

Gets marimba's allowable clefs.

```
>>> marimba.allowable_clefs
ClefInventory([Clef(name='treble'), Clef(name='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(contents=('Marimba',))
```

```
>>> show(marimba.instrument_name_markup)
```

Marimba

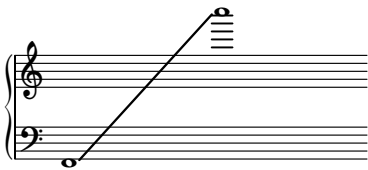
Returns markup.

`Marimba.pitch_range`

Gets marimba's range.

```
>>> marimba.pitch_range
PitchRange(range_string='[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(contents=('Mb.',))
```

```
>>> show(marimba.short_instrument_name_markup)
```

Mb.

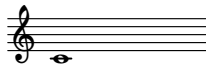
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

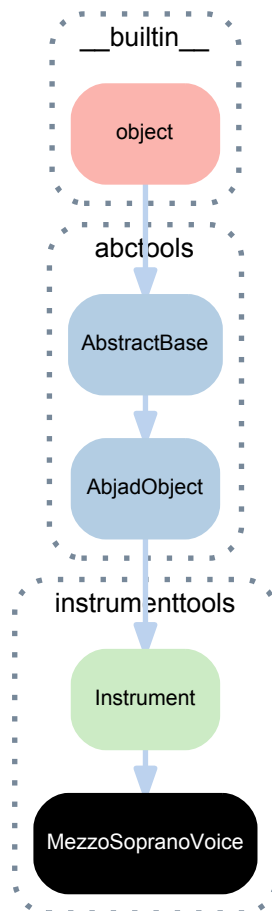
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

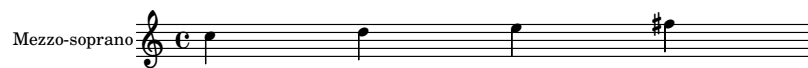
5.1.33 instrumenttools.MezzoSopranoVoice



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

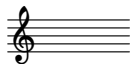
Read-only properties

MezzoSopranoVoice.allowable_clefs

Gets mezzo-soprano's allowable clefs.

```
>>> mezzo_soprano.allowable_clefs
ClefInventory([Clef(name='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(contents=('Mezzo-soprano',))
```

```
>>> show(mezzo_soprano.instrument_name_markup)
```

Mezzo-soprano

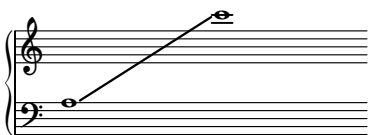
Returns markup.

MezzoSopranoVoice.pitch_range

Gets mezzo-soprano's range.

```
>>> mezzo_soprano.pitch_range
PitchRange(range_string='[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(contents=('Mezz.',))
```

```
>>> show(mezzo_soprano.short_instrument_name_markup)
```

Mezz.

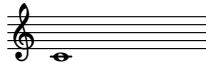
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

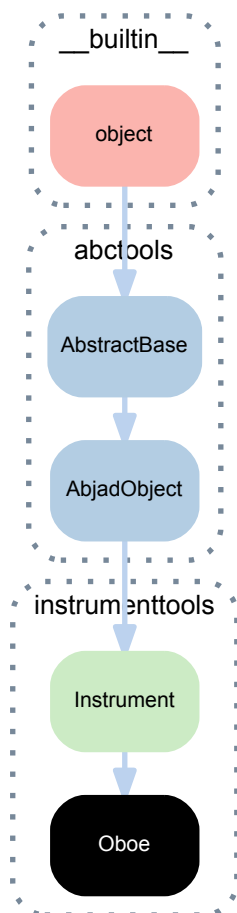
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.34 instrumenttools.Oboe

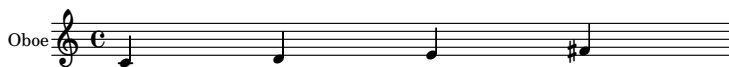


class instrumenttools.Oboe (*instrument_name='oboe', short_instrument_name='ob.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[Bb3, A6]', sounding_pitch_of_written_middle_c=None*)

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
- abctools.AbjadObject.AbstractBase
- __builtin__.object

Read-only properties

Oboe.**allowable_clefs**
Gets oboe's allowable clefs.

```
>>> oboe.allowable_clefs
ClefInventory ([Clef (name='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 (contents=('Oboe',))
```

```
>>> show(oboe.instrument_name_markup)
```

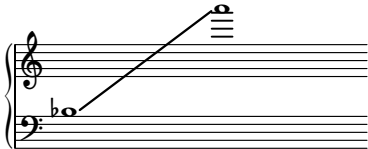
Oboe

Returns markup.

Oboe.**pitch_range**
Gets oboe's range.

```
>>> oboe.pitch_range
PitchRange (range_string=' [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 (contents=('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.

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

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

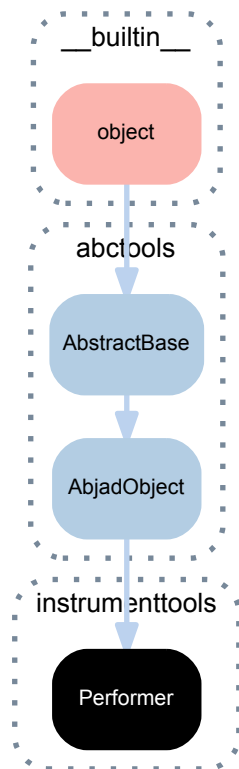
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.35 instrumenttools.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(
          contents=('Flute',),
        ),
        short_instrument_name_markup=markuptools.Markup(
          contents=('Fl.',),
        ),
        allowable_clefs=indicatortools.ClefInventory(
          [
            indicatortools.Clef(
              name='treble',
            ),
          ]
        ),
        pitch_range=pitchtools.PitchRange(
          range_string='[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(
          contents=('Piccolo',),
        ),
      ),
    ]
  )
)
```

```

short_instrument_name_markup=markuptools.Markup(
    contents=('Picc.',),
),
allowable_clefs=indicatortools.ClefInventory(
    [
        indicatortools.Clef(
            name='treble',
        ),
    ]
),
pitch_range=pitchtools.PitchRange(
    range_string='[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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`Performer.instrument_count`

Number of instruments to be played by performer.

```
>>> performer.instrument_count
2
```

Returns nonnegative integer

`Performer.is_doubling`

Is 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.get_instrument(*instrument_name*)

Gets instrument in performer with *instrument_name*.

```
>>> flutist = instrumenttools.Performer(name='flutist')
>>> flutist.instruments.append(instrumenttools.Flute())
>>> flutist.instruments.append(instrumenttools.Piccolo())
>>> flutist.get_instrument('piccolo')
Piccolo()
```

Returns instrument or none.

Performer.make_performer_name_instrument_dictionary()

Makes performer name / instrument dictionary.

```
>>> dictionary = \
...     performer.make_performer_name_instrument_dictionary()
>>> for key, value in sorted(dictionary.items()):
...     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
oboist:
  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)`

Is 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.

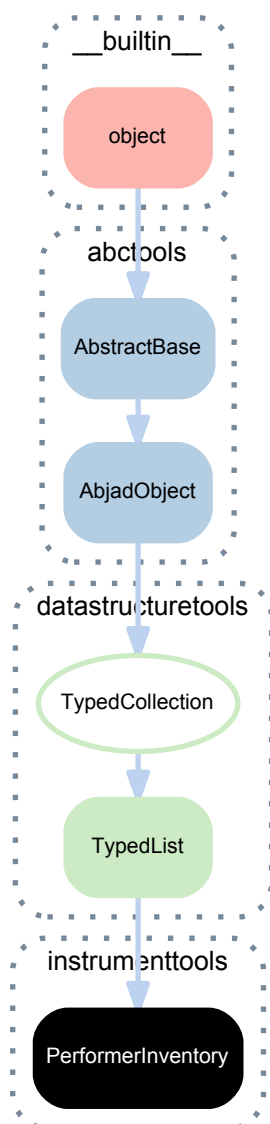
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

5.1.36 instrumenttools.PerformerInventory



class `instrumenttools.PerformerInventory` (*items=None*, *item_class=None*,
keep_sorted=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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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.

`PerformerInventory.get_instrument(instrument_name)`
Gets first instrument in performer inventory with *instrument_name*.

```
>>> flutist = instrumenttools.Performer(name='flutist')
>>> flutist.instruments.append(instrumenttools.Flute())
>>> flutist.instruments.append(instrumenttools.Piccolo())
>>> inventory = instrumenttools.PerformerInventory(
...     [flutist],
...     )
>>> inventory.get_instrument('piccolo')
Piccolo()
```

Returns instrument or none.

`(TypedList).index(item)`
Changes *item* to item and returns index.

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

Returns index.

`(TypedList).insert(i, item)`
Changes *item* 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(item)`
Changes *item* 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__(item)`
 Is true when typed collection container *item*. Otherwise false.
 Returns boolean.

`(TypedList).__delitem__(i)`
 Aliases `list.__delitem__()`.
 Returns none.

`(TypedCollection).__eq__(expr)`
 Is 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__()`.
 Returns item.

`(TypedCollection).__hash__()`
 Hashes typed collection.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 Returns integer.

`(TypedList).__iadd__(expr)`
 Changes items 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(
        name='ppp',
    ),
    indicatortools.Dynamic(
        name='p',
    ),
    indicatortools.Dynamic(
        name='mp',
    ),
    indicatortools.Dynamic(
        name='mf',
    ),
    indicatortools.Dynamic(
        name='fff',
    ),
],
```

```

    item_class=indicatortools.Dynamic,
)

```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__**().

Returns generator.

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

Changes items 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,
)

```

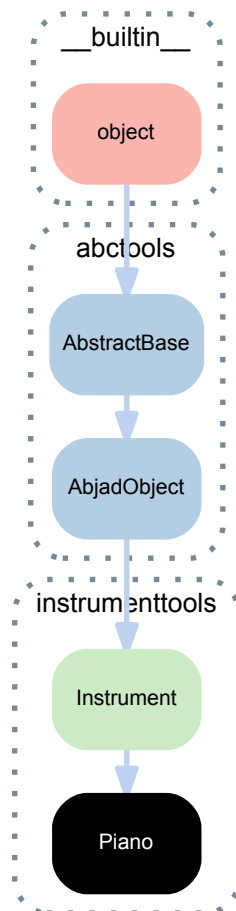
```

>>> 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,
)

```

Returns none.

5.1.37 instrumenttools.Piano

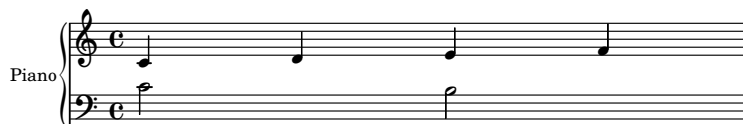


class `instrumenttools.Piano` (*instrument_name='piano', short_instrument_name='pf', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('treble', 'bass'), pitch_range='[A0, C8]', sounding_pitch_of_written_middle_c=None*)

A piano.

```

>>> staff_group = StaffGroup()
>>> staff_group.context_name = 'PianoStaff'
>>> staff_group.append(Staff("c'4 d'4 e'4 f'4"))
>>> staff_group.append(Staff("c'2 b2"))
>>> piano = instrumenttools.Piano()
>>> attach(piano, staff_group)
>>> attach(Clef(name='bass'), staff_group[1])
>>> show(staff_group)
  
```



The piano targets piano staff context by default.

Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`

- `__builtin__.object`

Read-only properties

`Piano.allowable_clefs`

Gets piano's allowable clefs.

```
>>> piano.allowable_clefs
ClefInventory([Clef(name='treble'), Clef(name='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(contents=('Piano',))
```

```
>>> show(piano.instrument_name_markup)
```

Piano

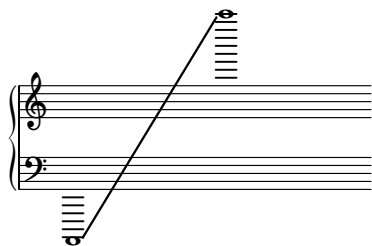
Returns markup.

`Piano.pitch_range`

Gets piano's range.

```
>>> piano.pitch_range
PitchRange(range_string='[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(contents=('Pf.',))
```

```
>>> show(piano.short_instrument_name_markup)
```

Pf.

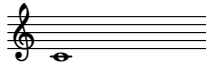
Returns markup.

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.

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

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

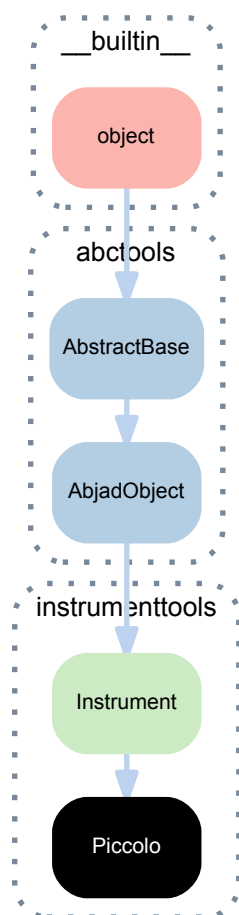
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.38 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')

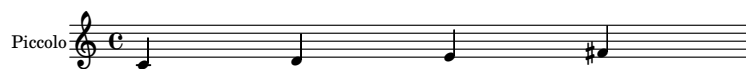
```

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

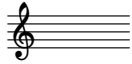
Read-only properties

Piccolo.allowable_clefs

Gets piccolo's allowable clefs.

```
>>> piccolo.allowable_clefs
ClefInventory([Clef(name='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(contents=('Piccolo',))
```

```
>>> show(piccolo.instrument_name_markup)
```

Piccolo

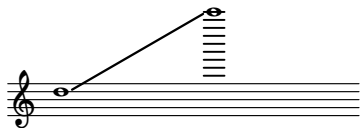
Returns markup.

Piccolo.pitch_range

Gets piccolo's range.

```
>>> piccolo.pitch_range
PitchRange(range_string='[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(contents=('Picc.',))
```

```
>>> show(piccolo.short_instrument_name_markup)
```

Picc.

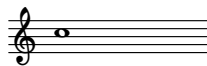
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

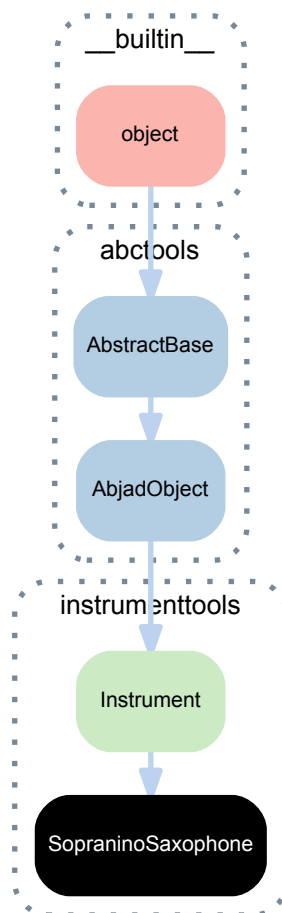
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.39 instrumenttools.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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

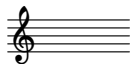
Read-only properties

SopraninoSaxophone.allowable_clefs

Gets sopranino saxophone's allowable clefs.

```
>>> sopranino_saxophone.allowable_clefs
ClefInventory([Clef(name='treble')])
```

```
>>> show(sopranino_saxophone.allowable_clefs)
```



Returns clef inventory.

SopraninoSaxophone.instrument_name

Gets sopranino saxophone's name.

```
>>> sopranino_saxophone.instrument_name
'sopranino saxophone'
```

Returns string.

SopraninoSaxophone.instrument_name_markup

Gets sopranino saxophone's instrument name markup.

```
>>> sopranino_saxophone.instrument_name_markup
Markup(contents=('Sopranino saxophone',))
```

```
>>> show(sopranino_saxophone.instrument_name_markup)
```

Sopranino saxophone

Returns markup.

SopraninoSaxophone.pitch_range

Gets sopranino saxophone's range.

```
>>> sopranino_saxophone.pitch_range
PitchRange(range_string='[Db4, F#6]')
```

```
>>> show(sopranino_saxophone.pitch_range)
```



Returns pitch range.

SopraninoSaxophone.short_instrument_name

Gets sopranino saxophone's short instrument name.

```
>>> sopranino_saxophone.short_instrument_name
'sopranino sax.'
```

Returns string.

SopraninoSaxophone.short_instrument_name_markup

Gets sopranino saxophone's short instrument name markup.

```
>>> sopranino_saxophone.short_instrument_name_markup
Markup(contents=('Sopranino sax.',))
```

```
>>> show(sopranino_saxophone.short_instrument_name_markup)
```

Sopranino sax.

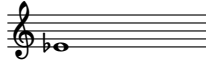
Returns markup.

SopraninoSaxophone.**sounding_pitch_of_written_middle_c**

Gets sounding pitch of sopranino 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.

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

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

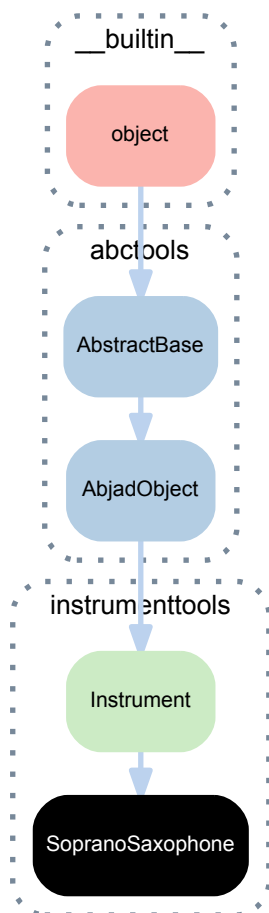
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.40 instrumenttools.SopranoSaxophone



```

class instrumenttools.SopranoSaxophone (instrument_name='soprano saxo-
phone', short_instrument_name='sop. sax.', instrument_name_markup=None,
short_instrument_name_markup=None, allow-
able_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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

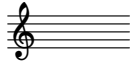
Read-only properties

SopranoSaxophone.allowable_clefs

Gets soprano saxophone's allowable clefs.

```
>>> soprano_saxophone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Soprano saxophone',))
```

```
>>> show(soprano_saxophone.instrument_name_markup)
```

Soprano saxophone

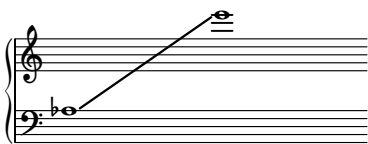
Returns markup.

SopranoSaxophone.pitch_range

Gets soprano saxophone's range.

```
>>> soprano_saxophone.pitch_range
PitchRange(range_string='[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(contents=('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.

`(AbjadObject).__ne__(expr)`

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

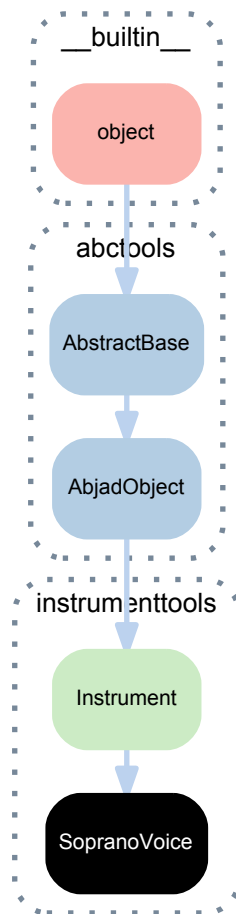
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.41 instrumenttools.SopranoVoice



```

class instrumenttools.SopranoVoice (instrument_name='soprano',
                                     short_instrument_name='sop.',
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,
                                     allowable_clefs=None,
                                     pitch_range='[C4, E6]',
                                     sounding_pitch_of_written_middle_c=None)

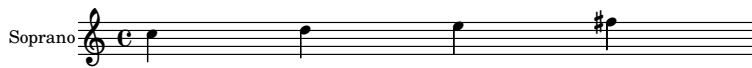
```

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

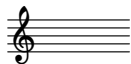
Read-only properties

SopranoVoice.allowable_clefs

Gets soprano's allowable clefs.

```
>>> soprano.allowable_clefs
ClefInventory([Clef(name='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(contents=('Soprano',))
```

```
>>> show(soprano.instrument_name_markup)
```

Soprano

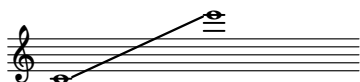
Returns markup.

SopranoVoice.pitch_range

Gets soprano's range.

```
>>> soprano.pitch_range
PitchRange(range_string='[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(contents=('Sop.',))
```

```
>>> show(soprano.short_instrument_name_markup)
```

Sop.

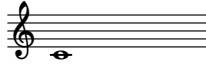
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

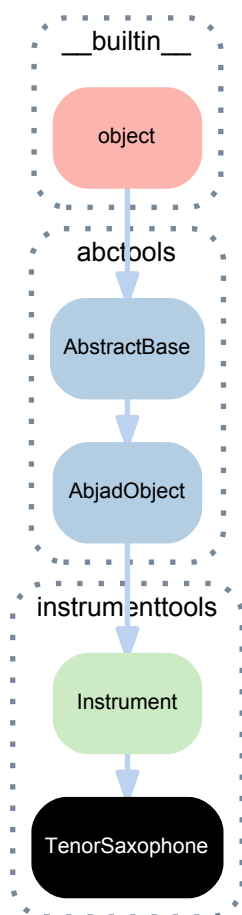
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.42 instrumenttools.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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`TenorSaxophone.allowable_clefs`

Gets tenor saxophone's allowable clefs.

```
>>> tenor_saxophone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Tenor saxophone',))
```

```
>>> show(tenor_saxophone.instrument_name_markup)
```

Tenor saxophone

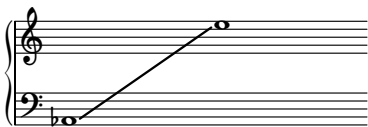
Returns markup.

`TenorSaxophone.pitch_range`

Gets tenor saxophone's range.

```
>>> tenor_saxophone.pitch_range
PitchRange(range_string='[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(contents=('Ten. sax.',))
```

```
>>> show(tenor_saxophone.short_instrument_name_markup)
```

Ten. sax.

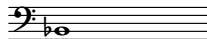
Returns markup.

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.

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

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

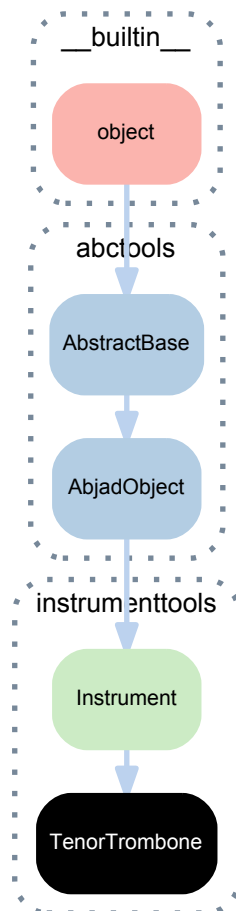
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.43 instrumenttools.TenorTrombone



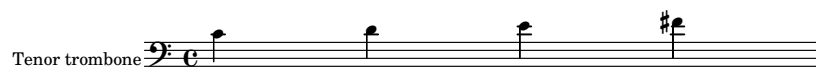
```

class instrumenttools.TenorTrombone (instrument_name='tenor',          trombone',
                                     short_instrument_name='ten.',    trb.',
                                     instrument_name_markup=None,
                                     short_instrument_name_markup=None,
                                     allowable_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(name='bass')
>>> attach(clef, staff)
>>> tenor_trombone = instrumenttools.TenorTrombone()
>>> attach(tenor_trombone, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

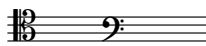
Read-only properties

`TenorTrombone.allowable_clefs`

Gets tenor trombone's allowable clefs.

```
>>> tenor_trombone.allowable_clefs
ClefInventory([Clef(name='tenor'), Clef(name='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(contents=('Tenor trombone',))
```

```
>>> show(tenor_trombone.instrument_name_markup)
```

Tenor trombone

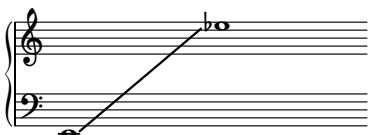
Returns markup.

`TenorTrombone.pitch_range`

Gets tenor trombone's range.

```
>>> tenor_trombone.pitch_range
PitchRange(range_string='[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(contents=('Ten. trb.',))
```

```
>>> show(tenor_trombone.short_instrument_name_markup)
```

Ten. trb.

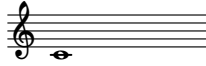
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

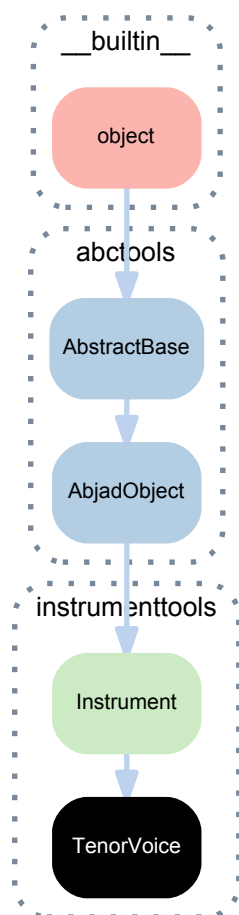
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.44 instrumenttools.TenorVoice

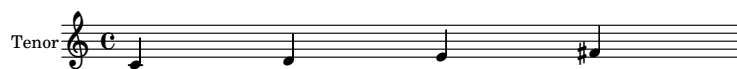


class `instrumenttools.TenorVoice` (*instrument_name='tenor', short_instrument_name='ten.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, pitch_range='[C3, D5]', sounding_pitch_of_written_middle_c=None*)

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

Read-only properties

TenorVoice.allowable_clefs

Gets tenor's allowable clefs.

```
>>> tenor.allowable_clefs
ClefInventory([Clef(name='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(contents=('Tenor',))
```

```
>>> show(tenor.instrument_name_markup)
```

Tenor

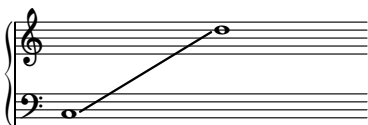
Returns markup.

TenorVoice.pitch_range

Gets tenor's range.

```
>>> tenor.pitch_range
PitchRange(range_string='[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(contents=('Ten.',))
```

```
>>> show(tenor.short_instrument_name_markup)
```

Ten.

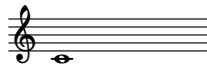
Returns markup.

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.

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

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

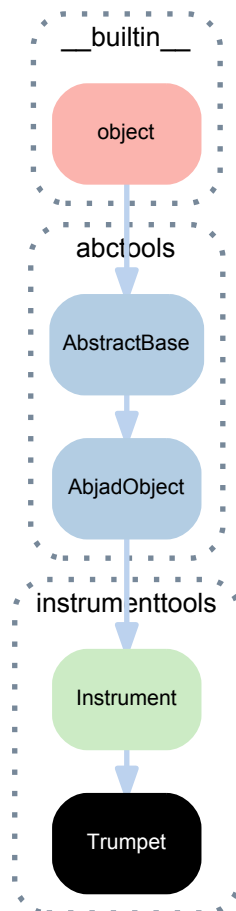
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.45 instrumenttools.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)

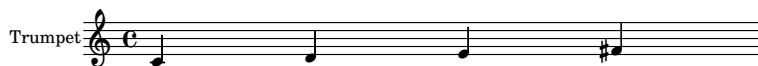
```

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

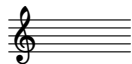
Read-only properties

Trumpet.**allowable_clefs**

Gets trumpet's allowable clefs.

```
>>> trumpet.allowable_clefs
ClefInventory([Clef(name='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(contents=('Trumpet',))
```

```
>>> show(trumpet.instrument_name_markup)
```

Trumpet

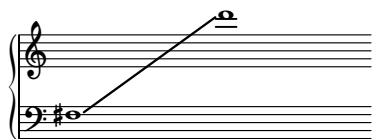
Returns markup.

Trumpet.**pitch_range**

Gets trumpet's range.

```
>>> trumpet.pitch_range
PitchRange(range_string='[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(contents=('Tp.',))
```

```
>>> show(trumpet.short_instrument_name_markup)
```

Tp.

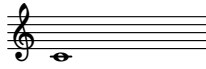
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

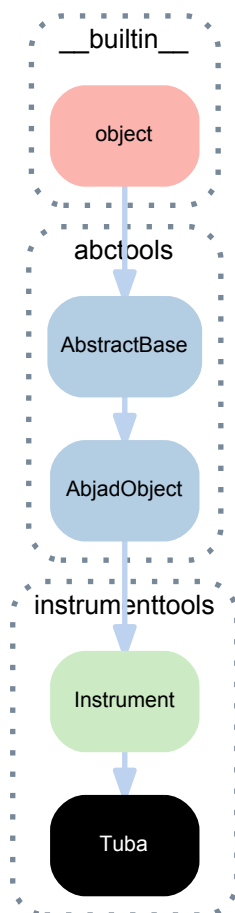
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.46 instrumenttools.Tuba

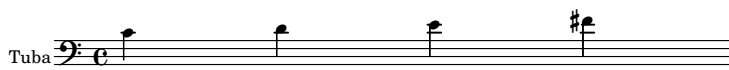


class `instrumenttools.Tuba` (*instrument_name='tuba', short_instrument_name='tb.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('bass',), pitch_range='[D1, F4]', sounding_pitch_of_written_middle_c=None*)

A tuba.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef(name='bass')
>>> attach(clef, staff)
>>> tuba = instrumenttools.Tuba()
>>> attach(tuba, staff)
>>> show(staff)
  
```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

Tuba.**allowable_clefs**

Gets tuba's allowable clefs.

```
>>> tuba.allowable_clefs
ClefInventory([Clef(name='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(contents=('Tuba',))
```

```
>>> show(tuba.instrument_name_markup)
```

Tuba

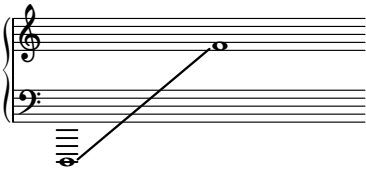
Returns markup.

Tuba.**pitch_range**

Gets tuba's range.

```
>>> tuba.pitch_range
PitchRange(range_string='[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(contents=('Tb.',))
```

```
>>> show(tuba.short_instrument_name_markup)
```

Tb.

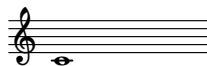
Returns markup.

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.

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

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

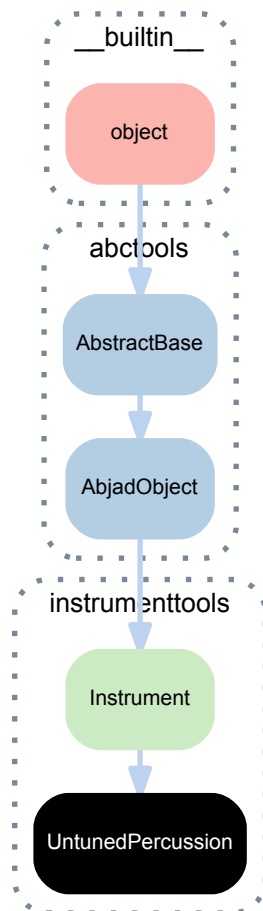
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.47 instrumenttools.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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

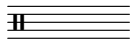
Read-only properties

`UntunedPercussion.allowable_clefs`

Gets untuned percussion's allowable clefs.

```
>>> untuned_percussion.allowable_clefs
ClefInventory([Clef(name='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(contents=('Untuned percussion',))
```

```
>>> show(untuned_percussion.instrument_name_markup)
```

Untuned percussion

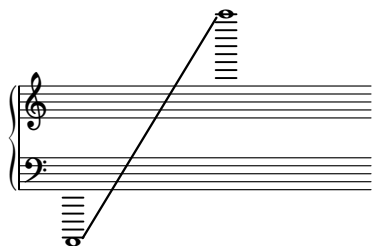
Returns markup.

`UntunedPercussion.pitch_range`

Gets untuned percussion's range.

```
>>> untuned_percussion.pitch_range
PitchRange(range_string='[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(contents=('Perc.',))
```

```
>>> show(untuned_percussion.short_instrument_name_markup)
```

Perc.

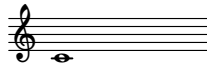
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

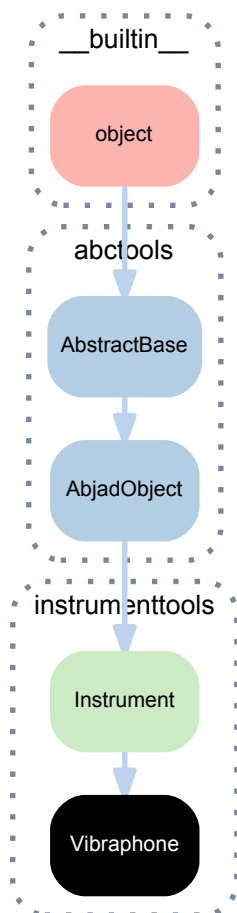
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.48 instrumenttools.Vibraphone



```

class instrumenttools.Vibraphone (instrument_name='vibraphone',
                                   short_instrument_name='vibr',
                                   strument_name_markup=None,
                                   short_instrument_name_markup=None,
                                   able_clefs=None,    pitch_range='[F3, F6]',
                                   ing_pitch_of_written_middle_c=None)

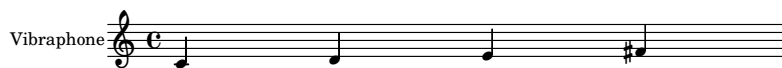
```

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

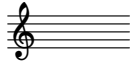
Read-only properties

Vibraphone.allowable_clefs

Gets vibraphone's allowable clefs.

```
>>> vibraphone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Vibraphone',))
```

```
>>> show(vibraphone.instrument_name_markup)
```

Vibraphone

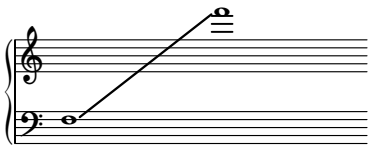
Returns markup.

Vibraphone.pitch_range

Gets vibraphone's range.

```
>>> vibraphone.pitch_range
PitchRange(range_string='[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(contents=('Vibr.',))
```

```
>>> show(vibraphone.short_instrument_name_markup)
```

Vibr.

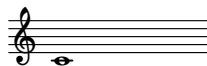
Returns markup.

`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.

`(AbjadObject).__ne__(expr)`

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

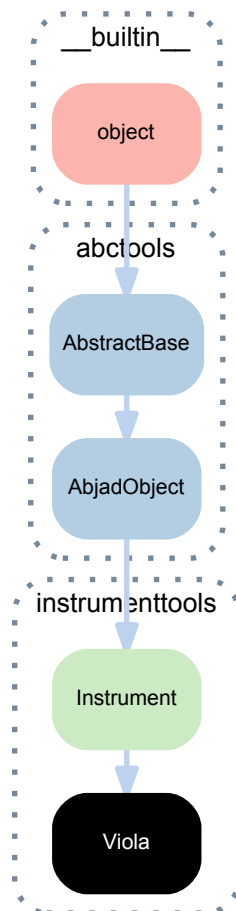
Returns boolean.

`(Instrument).__repr__()`

Gets interpreter representation of instrument.

Returns string.

5.1.49 instrumenttools.Viola



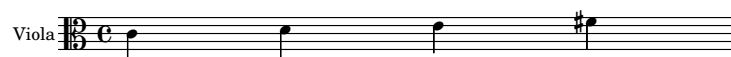
class `instrumenttools.Viola` (*instrument_name='viola', short_instrument_name='va', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=('alto', 'treble'), default_tuning=('C3', 'G3', 'D4', 'A4'), pitch_range='[C3, D6]', sounding_pitch_of_written_middle_c=None*)

A viola.

```

>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> clef = Clef(name='alto')
>>> attach(clef, staff)
>>> viola = instrumenttools.Viola()
>>> attach(viola, staff)
>>> show(staff)

```



Bases

- `instrumenttools.Instrument`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

Viola.allowable_clefs

Gets viola's allowable clefs.

```
>>> viola.allowable_clefs
ClefInventory([Clef(name='alto'), Clef(name='treble')])
```

```
>>> show(viola.allowable_clefs)
```



Returns clef inventory.

Viola.default_tuning

Gets viola's default tuning.

```
>>> viola.default_tuning
Tuning(pitches=PitchSegment(['c', 'g', 'd', 'a']))
```

Returns tuning.

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(contents=('Viola',))
```

```
>>> show(viola.instrument_name_markup)
```

Viola

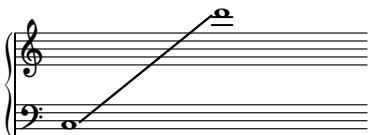
Returns markup.

Viola.pitch_range

Gets viola's range.

```
>>> viola.pitch_range
PitchRange(range_string='[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(contents=('Va.',))
```

```
>>> show(viola.short_instrument_name_markup)
```

Va.

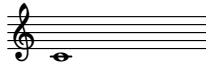
Returns markup.

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.

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

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

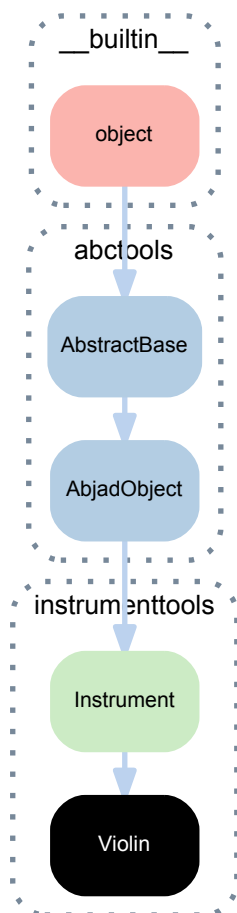
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

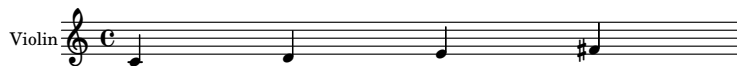
5.1.50 instrumenttools.Violin



class `instrumenttools.Violin` (*instrument_name='violin', short_instrument_name='vn.', instrument_name_markup=None, short_instrument_name_markup=None, allowable_clefs=None, default_tuning=('G3', 'D4', 'A4', 'E5'), 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

Violin.allowable_clefs

Gets violin's allowable clefs.

```
>>> violin.allowable_clefs
ClefInventory([Clef(name='treble')])
```

```
>>> show(violin.allowable_clefs)
```



Returns clef inventory.

Violin.default_tuning

Gets violin's default tuning.

```
>>> violin.default_tuning
Tuning(pitches=PitchSegment(['g', 'd', 'a', 'e']))
```

Returns tuning.

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(contents=('Violin',))
```

```
>>> show(violin.instrument_name_markup)
```

Violin

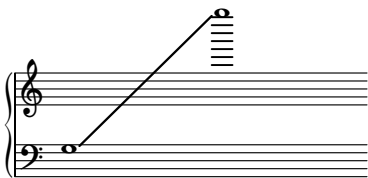
Returns markup.

Violin.pitch_range

Gets violin's range.

```
>>> violin.pitch_range
PitchRange(range_string='[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(contents=('Vn.',))
```

```
>>> show(violin.short_instrument_name_markup)
```

Vn.

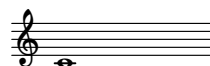
Returns markup.

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.

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

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

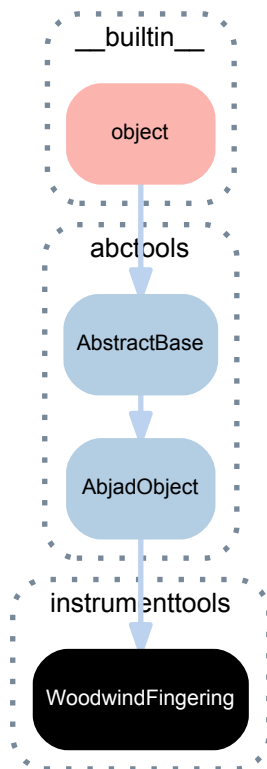
Returns boolean.

(Instrument).**__repr__**()

Gets interpreter representation of instrument.

Returns string.

5.1.51 instrumenttools.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:

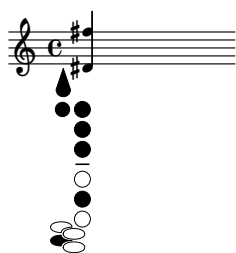
```
>>> woodwind_fingering_2 = instrumenttools.WoodwindFingering(
...     woodwind_fingering)
>>> print(format(woodwind_fingering_2))
instrumenttools.WoodwindFingering(
    instrument_name='clarinet',
    center_column=('one', 'two', 'three', 'five'),
    left_hand=('R', 'thumb'),
    right_hand=('e',),
)
```

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(contents=fingering_command, direction=Down)
>>> chord = Chord("<ds' fs''>4")
>>> attach(markup, chord)
>>> show(chord)
```



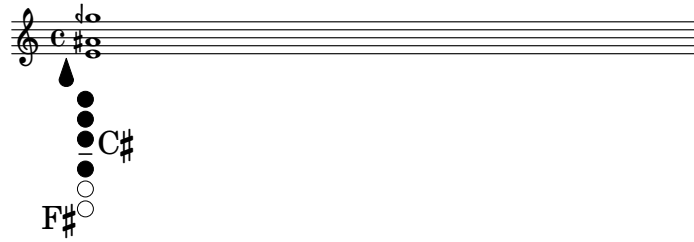
Initializes fingerings for eight different woodwind instruments:

```
>>> 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:

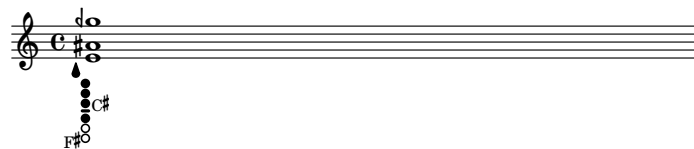
```
>>> chord = Chord("<e' as' gqf''>1")
>>> fingering = instrumenttools.WoodwindFingering(
...     'clarinet',
...     center_column=['one', 'two', 'three', 'four'],
...     left_hand=['R', 'cis'],
...     right_hand=['fis'])
>>> diagram = fingering()
>>> not_graphical = markuptools.MarkupCommand(
...     'override',
...     schemetools.SchemePair('graphical', False))
>>> markup = markuptools.Markup(contents=
```

```
... [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' 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))
>>> size = markuptools.MarkupCommand(
...     'override', schemetools.SchemePair('size', .5))
>>> thickness = markuptools.MarkupCommand(
...     'override', schemetools.SchemePair('thickness', .4))
>>> markup = markuptools.Markup(contents=
...     [not_graphical, size, thickness, diagram], direction=Down)
>>> attach(markup, chord)
>>> show(chord)
```



Inspired by Mike Solomon's LilyPond woodwind diagrams.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

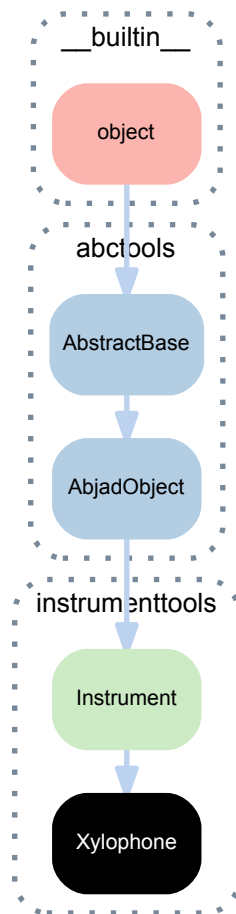
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

5.1.52 instrumenttools.Xylophone



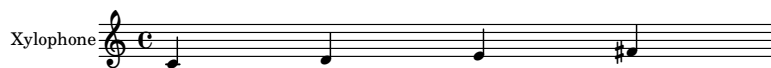
```

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

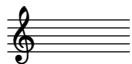
Read-only properties

`Xylophone.allowable_clefs`

Gets xylophone's allowable clefs.

```
>>> xylophone.allowable_clefs
ClefInventory([Clef(name='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(contents=('Xylophone',))
```

```
>>> show(xylophone.instrument_name_markup)
```

Xylophone

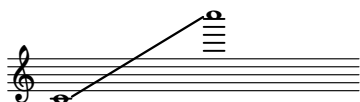
Returns markup.

`Xylophone.pitch_range`

Gets xylophone's range.

```
>>> xylophone.pitch_range
PitchRange(range_string='[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(contents=('Xyl.',))
```

```
>>> show(xylophone.short_instrument_name_markup)
```

Xyl.

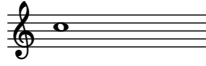
Returns markup.

`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.

`(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*)

Is 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*)

Is true when notes and chords in *expr* are on expected clefs.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> clef = Clef(name='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(name='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(name='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(2 * Measure((2, 8), "c'8 d'8"))
>>> scoretools.scale_measure_denominator_and_adjust_measure_contents(staff[1], 3)
Measure((3, 12), "c'8. d'8.", implicit_scaling=True)
```

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



```
>>> labeltools.color_measures_with_non_power_of_two_denominators_in_expr(staff, 'red')
[Measure((3, 12), "c'8. d'8.", implicit_scaling=True)]
```

```
>>> 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.Staff([tuplet])
>>> staff.context_name = 'RhythmicStaff'
>>> 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.Staff([tuplet])
>>> staff.context_name = 'RhythmicStaff'
>>> 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.Staff([tuplet])
>>> staff.context_name = 'RhythmicStaff'
>>> 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.

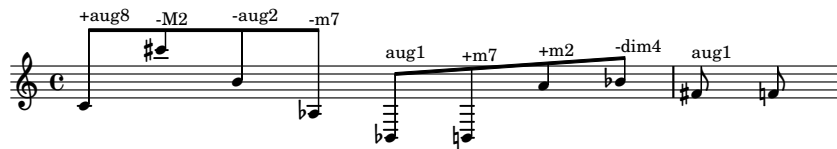
6.1.13 `labeltools.label_leaves_in_expr_with_named_interval_classes`

`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.

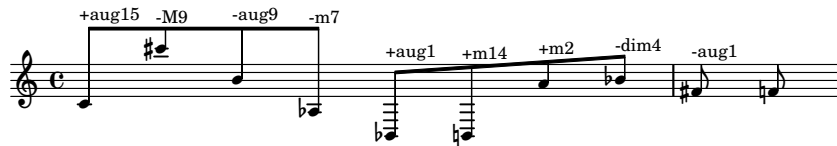
6.1.14 `labeltools.label_leaves_in_expr_with_named_intervals`

`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.

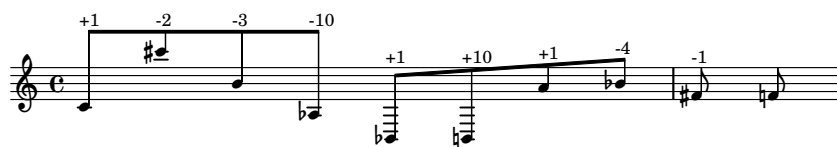
6.1.15 `labeltools.label_leaves_in_expr_with_numbered_interval_classes`

`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.

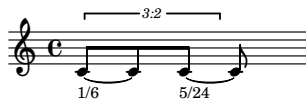
6.1.22 labeltools.label_logical_ties_in_expr_with_logical_tie_duration

`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.

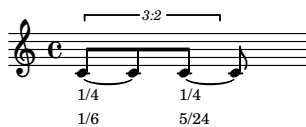
6.1.23 `labeltools.label_logical_ties_in_expr_with_logical_tie_durations`

`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.

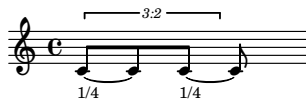
6.1.24 `labeltools.label_logical_ties_in_expr_with_written_logical_tie_duration`

`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`

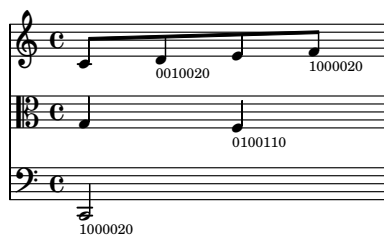
`labeltools.label_vertical_moments_in_expr_with_interval_class_vectors` (*expr*,
markup_direction=Down)

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`

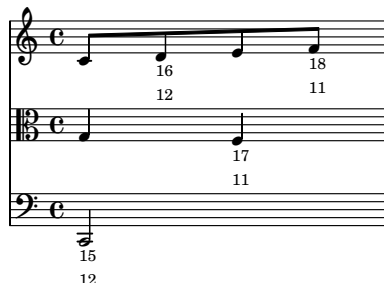
`labeltools.label_vertical_moments_in_expr_with_named_intervals` (*expr*,
markup_direction=Down)

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`

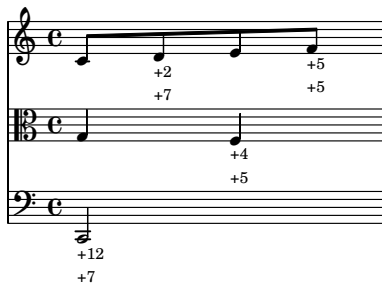
`labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes` (*expr*,
markup_direction=Down)

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`

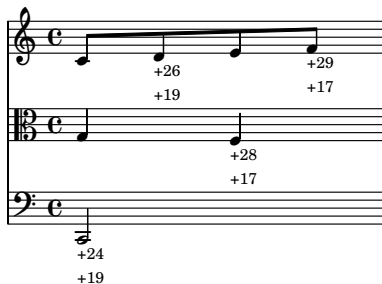
`labeltools.label_vertical_moments_in_expr_with_numbered_intervals` (*expr*,
markup_direction=Down)

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
}
```

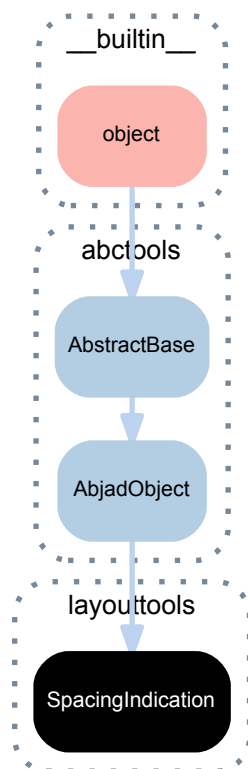
```
>>> show(staff)
```



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=Duration(1, 8), units_per_minute=44), Duration(1, 68))
```

Initialize from constants:

```
>>> layouttools.SpacingIndication((1, 8), 44, (1, 68))
SpacingIndication(Tempo(duration=Duration(1, 8), units_per_minute=44), Duration(1, 68))
```

Initialize from other spacing indication:

```
>>> layouttools.SpacingIndication(indication)
SpacingIndication(Tempo(duration=Duration(1, 8), units_per_minute=44), Duration(1, 68))
```

Spacing indications are immutable.

Bases

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

Read-only properties

`SpacingIndication.normalized_spacing_duration`

Proportional notation duration normalized to 60 MM.

Returns duration.

`SpacingIndication.proportional_notation_duration`

LilyPond proportional notation duration of spacing indication.

Returns duration.

`SpacingIndication.tempo_indication`

Tempo of spacing indication.

Returns tempo.

Special methods

`SpacingIndication.__eq__(expr)`

Spacing indications compare equal when normalized spacing durations compare equal.

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

Formats Abjad object.

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

Returns string.

`SpacingIndication.__hash__()`

Hashes spacing indication.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

7.2 Functions

7.2.1 layouttools.make_spacing_vector

`layouttools.make_spacing_vector` (*basic_distance*, *minimum_distance*, *padding*, *stretchability*)

Makes 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_emptyBars=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_by_line_duration_ge

`layouttools.set_line_breaks_by_line_duration_ge` (*expr*, *line_duration*,
line_break_class=None,
add_emptyBars=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_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_by_line_duration_in_seconds_ge

`layouttools.set_line_breaks_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_by_line_duration_in_seconds_ge(
...     staff, Duration(6))
>>> show(staff)
```



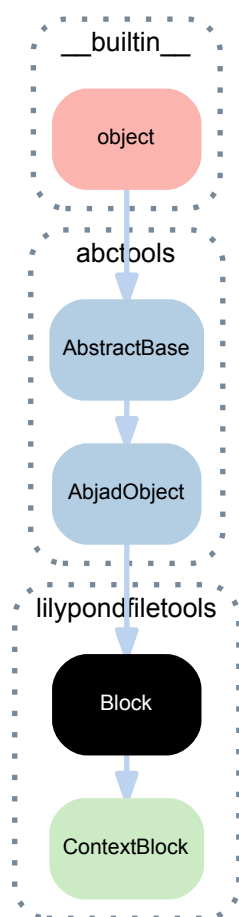
```
>>> print(format(staff))
\new Staff {
  \tempo 8=44
  {
    \time 2/8
    c'8
    d'8
  }
}
```

```
        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 Concrete classes

8.1.1 lilypondfiletools.Block



class lilypondfiletools.**Block** (*name='score'*)
A LilyPond file block.

```
>>> block = lilypondfiletools.Block(name='paper')
>>> block.left_margin = lilypondfiletools.LilyPondDimension(2, 'cm')
>>> block.right_margin = lilypondfiletools.LilyPondDimension(2, 'cm')
>>> block
<Block(name='paper')>
```

```
>>> print(format(block))
\paper {
  left-margin = 2\cm
```

```
right-margin = 2\cm
}
```

```
>>> block = lilypondfiletools.Block(name='score')
>>> markup = Markup('foo')
>>> block.items.append(markup)
>>> block
<Block(name='score')>
```

```
>>> print(format(block))
\score {
  {
    \markup { foo }
  }
}
```

Bases

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

Read-only properties

`Block.items`

Gets items in block.

```
>>> block = lilypondfiletools.Block(name='score')
>>> markup = Markup('foo')
>>> block.items.append(markup)
```

```
>>> block.items
[Markup(contents=('foo',))]
```

Returns list.

`Block.name`

Gets name of block.

```
>>> block = lilypondfiletools.Block(name='score')
>>> markup = Markup('foo')
>>> block.items.append(markup)
```

```
>>> block.name
'score'
```

Returns string.

Special methods

`(AbjadObject).__eq__(expr)`

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

Returns boolean.

`Block.__format__(format_specification='')`

Formats block.

Returns string.

`Block.__getitem__(name)`

Gets block item with *name*.

Gets score with name 'Red Example Score' in score block:

```
>>> block = lilypondfiletools.Block(name='score')
>>> score = Score(name='Red Example Score')
>>> block.items.append(score)
```

```
>>> block['Red Example Score']
Score(is_simultaneous=True)
```

Raises key error when no item with *name* is found.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

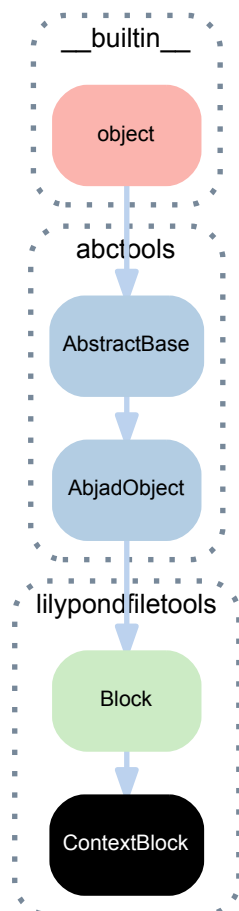
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

8.1.2 lilypondfiletools.ContextBlock



```
class lilypondfiletools.ContextBlock (source_context_name=None,          name=None,
                                     type_=None, alias=None)
```

A LilyPond file \context block.

```
>>> block = lilypondfiletools.ContextBlock(
...     source_context_name='Staff',
...     name='FluteStaff',
...     type_='Engraver_group',
...     alias='Staff',
... )
>>> block.remove_commands.append('Forbid_line_break_engraver')
>>> block.consists_commands.append('Horizontal_bracket_engraver')
>>> block.accepts_commands.append('FluteUpperVoice')
>>> block.accepts_commands.append('FluteLowerVoice')
>>> override(block).beam.positions = (-4, -4)
>>> override(block).stem.stem_end_position = -6
>>> set_(block).auto_beaming = False
>>> set_(block).tuplet_full_length = True
>>> block
<ContextBlock(source_context_name='Staff', name='FluteStaff', type_='Engraver_group', alias='Staff')>
```

```
>>> print(format(block))
\context {
  \Staff
  \name FluteStaff
  \type Engraver_group
  \alias Staff
  \remove Forbid_line_break_engraver
  \consists Horizontal_bracket_engraver
  \accepts FluteUpperVoice
  \accepts FluteLowerVoice
  \override Beam #'positions = #'(-4 . -4)
  \override Stem #'stem-end-position = #-6
  autoBeaming = ##f
  tupletFullLength = ##t
}
```

Bases

- `lilypondfiletools.Block`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`ContextBlock.accepts_commands`

Gets arguments of LilyPond \accepts commands.

```
>>> block.accepts_commands
['FluteUpperVoice', 'FluteLowerVoice']
```

Returns list.

`ContextBlock.alias`

Gets and sets argument of LilyPond \alias command.

```
>>> block.alias
'Staff'
```

Returns string or none.

`ContextBlock.consists_commands`

Gets arguments of LilyPond \consists commands.


```
>>> block.consists_commands
['Horizontal_bracket_engraver']
```

Returns list.

`ContextBlock.items`

Gets items in context block.

```
>>> block.items
[]
```

Returns list.

`ContextBlock.name`

Gets and sets argument of LilyPond `\name` command.

```
>>> block.name
'FluteStaff'
```

Returns string or none.

`ContextBlock.remove_commands`

Gets arguments of LilyPond `\remove` commands.

```
>>> block.remove_commands
['Forbid_line_break_engraver']
```

Returns list.

`ContextBlock.source_context_name`

Gets and sets source context name.

```
>>> block.source_context_name
'Staff'
```

Returns string or none.

`ContextBlock.type_`

Gets and sets argument of LilyPond `\type` command.

```
>>> block.type_
'Engraver_group'
```

Returns string or none.

Special methods

`(AbjadObject).__eq__(expr)`

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

Returns boolean.

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

Formats block.

Returns string.

`(Block).__getitem__(name)`

Gets block item with *name*.

Gets score with name 'Red Example Score' in score block:

```
>>> block = lilypondfiletools.Block(name='score')
>>> score = Score(name='Red Example Score')
>>> block.items.append(score)
```

```
>>> block['Red Example Score']
Score(is_simultaneous=True)
```

Raises key error when no item with *name* is found.

(AbjadObject).**__hash__**()
Hashes Abjad object.

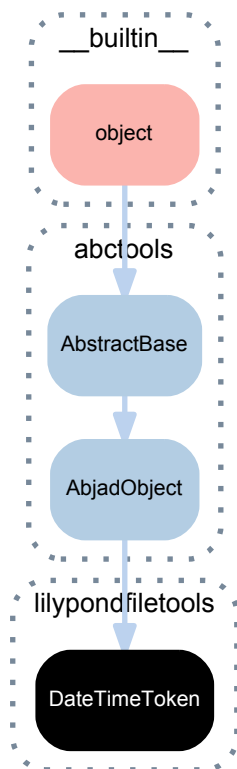
Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

8.1.3 lilypondfiletools.DateTimeToken



class lilypondfiletools.**DateTimeToken** (*date_string=None*)
A LilyPond file date / time token.

```
>>> lilypondfiletools.DateTimeToken()
DateTimeToken()
```

Bases

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

Read-only properties

`DateTimeToken.date_string`

Gets date string of date / time token.

```
>>> token = lilypondfiletools.DateTimeToken()
>>> token.date_string
'2014-01-23 12:21'
```

Returns string.

Special methods

`(AbjadObject).__eq__(expr)`

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

Returns boolean.

`DateTimeToken.__format__(format_specification='')`

Formats date / time token.

```
>>> token = lilypondfiletools.DateTimeToken()
>>> print(format(token))
2014-01-04 14:42
```

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

Returns boolean.

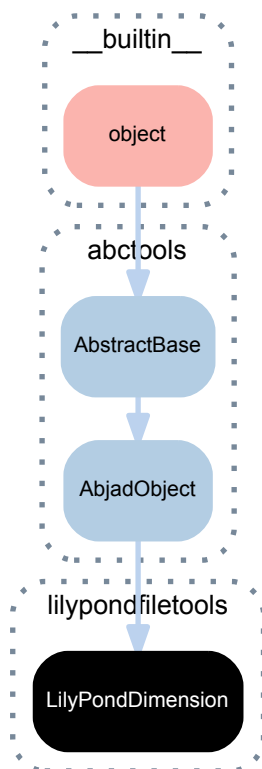
`DateTimeToken.__repr__()`

Gets interpreter representation of date / time token.

```
>>> lilypondfiletools.DateTimeToken()
DateTimeToken()
```

Returns string.

8.1.4 lilypondfiletools.LilyPondDimension



class lilypondfiletools.**LilyPondDimension** (*value=0, unit='cm'*)
 A LilyPond file \paper block dimension.

```
>>> lilypondfiletools.LilyPondDimension(2, 'in')
LilyPondDimension(value=2, unit='in')
```

Use for LilyPond file \paper block attributes.

Bases

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

Read-only properties

LilyPondDimension.unit
 Gets unit of LilyPond dimension.

```
>>> dimension = lilypondfiletools.LilyPondDimension(2, 'in')
>>> dimension.unit
'in'
```

Returns 'cm', 'in', 'mm' or 'pt'.

LilyPondDimension.value
 Gets value of LilyPond dimension.

```
>>> dimension = lilypondfiletools.LilyPondDimension(2, 'in')
>>> dimension.value
2
```

Returns number.

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.

```
>>> dimension = lilypondfiletools.LilyPondDimension(2, 'in')
>>> print(format(dimension))
2\in
```

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

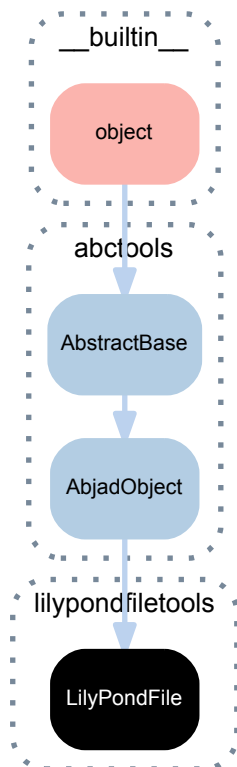
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

8.1.5 lilypondfiletools.LilyPondFile



class lilypondfiletools.**LilyPondFile**

A LilyPond file.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(staff)
>>> comment = 'File construct as an example.'
>>> lilypond_file.file_initial_user_comments.append(comment)
>>> comment = 'Parts shown here for positioning.'
>>> lilypond_file.file_initial_user_comments.append(comment)
>>> file_name = 'external-settings-file-1.ly'
>>> lilypond_file.file_initial_user_includes.append(file_name)
>>> file_name = 'external-settings-file-2.ly'
>>> lilypond_file.file_initial_user_includes.append(file_name)
>>> lilypond_file.default_paper_size = 'a5', 'portrait'
>>> lilypond_file.global_staff_size = 16
>>> lilypond_file.header_block.composer = Markup('Josquin')
>>> lilypond_file.header_block.title = Markup('Missa sexti tonus')
>>> lilypond_file.layout_block.indent = 0
>>> lilypond_file.layout_block.left_margin = 15
>>> lilypond_file
<LilyPondFile(4)>
```

```
>>> print(format(lilypond_file))
% 2004-01-14 17:29

% File construct as an example.
% Parts shown here for positioning.

\version "2.19.0"
\include "english.ly"

\include "external-settings-file-1.ly"
\include "external-settings-file-2.ly"

#(set-default-paper-size "a5" 'portrait)
#(set-global-staff-size 16)

\header {
  composer = \markup { Josquin }
  title = \markup { Missa sexti tonus }
}

\layout {
  indent = #0
  left-margin = #15
}

\paper {
}

\new Staff {
  c'8
  d'8
  e'8
  f'8
}
```

```
>>> show(lilypond_file)
```

Missa sexti tonus

Josquin



Bases

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

Read-only properties

LilyPondFile.file_initial_system_comments

Gets file-initial system comments of LilyPond file.

```
>>> for x in lilypond_file.file_initial_system_comments:
...     x
DateTimeToken('2014-01-04 17:16')
```

Returns list.

LilyPondFile.file_initial_system_includes

Gets file-initial system include commands of LilyPond file.

```
>>> for x in lilypond_file.file_initial_system_includes:
...     x
LilyPondVersionToken('2.19.11')
LilyPondLanguageToken('english')
```

Returns list.

LilyPondFile.file_initial_user_comments

Gets file-initial user comments of Lilypond file.

```
>>> for x in lilypond_file.file_initial_user_comments:
...     x
'File construct as an example.'
'Parts shown here for positioning.'
```

Returns list.

LilyPondFile.file_initial_user_includes

Gets file-initial user include commands of LilyPond file.

```
>>> for x in lilypond_file.file_initial_user_includes:
...     x
'external-settings-file-1.ly'
'external-settings-file-2.ly'
```

Returns list.

LilyPondFile.items

Gets items in LilyPond file.

```
>>> for item in lilypond_file.items:
...     item
<Block (name='header')>
<Block (name='layout')>
<Block (name='paper')>
<Block (name='score')>
```

Returns list.

Read/write properties

LilyPondFile.default_paper_size

Gets default paper size of LilyPond file.

```
>>> lilypond_file.default_paper_size
('a5', 'portrait')
```

Returns pair or none.

LilyPondFile.global_staff_size

Gets global staff size of LilyPond file.

```
>>> lilypond_file.global_staff_size
16
```

Returns number.

LilyPondFile.use_relative_includes
Gets boolean flag to use relative include paths.

```
>>> lilypond_file.use_relative_includes
False
```

Returns boolean.

Special methods

(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.

LilyPondFile.__getitem__(*name*)
Gets LilyPond file item with *name*.

```
>>> lilypond_file['header']
<Block(name='header')>
```

Raises key error when no item with *name* is found.

(AbjadObject).**__hash__**()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
Returns integer.

LilyPondFile.__illustrate__()
Illustrates LilyPond file.
Returns LilyPond file unchanged.

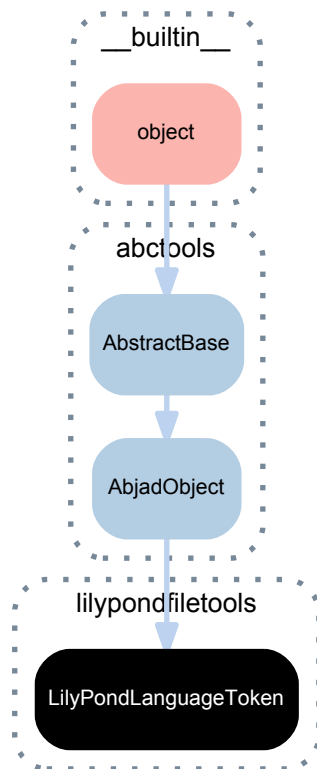
(AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

LilyPondFile.__repr__()
Gets interpreter representation of LilyPond file.

```
>>> lilypond_file
<LilyPondFile(4)>
```

Returns string.

8.1.6 lilypondfiletools.LilyPondLanguageToken



class lilypondfiletools.LilyPondLanguageToken
 A LilyPond file \language token.

```
>>> lilypondfiletools.LilyPondLanguageToken()
LilyPondLanguageToken('english')
```

Bases

- abctools.AbjadObject
- abctools.AbjadObject.AbstractBase
- __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.

```
>>> token = lilypondfiletools.LilyPondLanguageToken()
>>> print(format(token))
\language "english"
```

Returns string.

(AbjadObject).**__hash__**()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

Returns boolean.

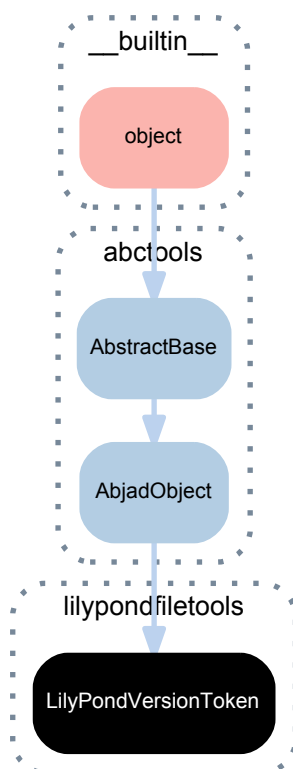
LilyPondLanguageToken.**__repr__**()

Gets interpreter representation of LilyPond language token.

```
>>> token = lilypondfiletools.LilyPondLanguageToken()
>>> token
LilyPondLanguageToken('english')
```

Returns string.

8.1.7 lilypondfiletools.LilyPondVersionToken



class lilypondfiletools.**LilyPondVersionToken**(*version_string=None*)
A LilyPond file \version token.

```
>>> lilypondfiletools.LilyPondVersionToken()
LilyPondVersionToken('2.19.0')
```

Bases

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

Read-only properties

`LilyPondVersionToken.version_string`

Gets version string of LilyPond version token.

Gets version string from install environment:

```
>>> token = lilypondfiletools.LilyPondVersionToken(
...     version_string=None,
... )
>>> token.version_string
'2.19.0'
```

Gets version string from explicit input:

```
>>> token = lilypondfiletools.LilyPondVersionToken(
...     version_string='2.19.0',
... )
>>> token.version_string
'2.19.0'
```

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.

```
>>> token = lilypondfiletools.LilyPondVersionToken()
>>> print(format(token))
\version "2.19.0"
```

Return string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

Returns boolean.

`LilyPondVersionToken.__repr__()`

Gets interpreter representation of LilyPond version_string token.

```
>>> token = lilypondfiletools.LilyPondVersionToken()
>>> token
LilyPondVersionToken('2.19.0')
```

Returns string.

8.2 Functions

8.2.1 `lilypondfiletools.make_basic_lilypond_file`

`lilypondfiletools.make_basic_lilypond_file(music=None)`

Makes basic LilyPond file.

```
>>> score = Score([Staff("c'8 d'8 e'8 f'8")])
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
>>> lilypond_file.header_block.title = Markup('Missa sexti tonus')
>>> lilypond_file.header_block.composer = Markup('Josquin')
>>> lilypond_file.layout_block.indent = 0
>>> lilypond_file.paper_block.top_margin = 15
>>> lilypond_file.paper_block.left_margin = 15
```

```
>>> print(format(lilypond_file))
\header {
  composer = \markup { Josquin }
  title = \markup { Missa sexti tonus }
}

\layout {
  indent = #0
}

\paper {
  left-margin = #15
  top-margin = #15
}

\score {
  \new Score <<
    \new Staff {
      c'8
      d'8
      e'8
      f'8
    }
  >>
}
```

```
>>> show(lilypond_file)
```

Missa sexti tonus

Josquin



Wraps *music* in LilyPond `\score` block.

Adds LilyPond `\header`, `\layout`, `\paper` and `\score` blocks to LilyPond file.

Returns LilyPond file.

8.2.2 lilypondfiletools.make_floating_time_signature_lilypond_file

`lilypondfiletools.make_floating_time_signature_lilypond_file` (*music=None*)

Makes floating time signature LilyPond file.

```
>>> score = Score()
>>> time_signature_context = scoretools.Context(
...     context_name='TimeSignatureContext',
... )
>>> durations = [(2, 8), (3, 8), (4, 8)]
>>> measures = scoretools.make_spacer_skip_measures(durations)
>>> time_signature_context.extend(measures)
>>> score.append(time_signature_context)
>>> staff = Staff()
>>> staff.append(Measure((2, 8), "c'8 ( d'8 )"))
>>> staff.append(Measure((3, 8), "e'8 ( f'8 g'8 )"))
>>> staff.append(Measure((4, 8), "fs'4 ( e'8 d'8 )"))
>>> score.append(staff)
>>> lilypond_file = \
...     lilypondfiletools.make_floating_time_signature_lilypond_file(
...     score
... )
```

```

>>> print(format(lilypond_file))
% 2014-01-07 18:22

\version "2.19.0"
\language "english"

#(set-default-paper-size "letter" 'portrait)
#(set-global-staff-size 12)

\header {}

\layout {
  \accidentalStyle forget
  indent = #0
  ragged-right = ##t
  \context {
    \name TimeSignatureContext
    \type Engraver_group
    \consists Axis_group_engraver
    \consists Time_signature_engraver
    \override TimeSignature #'X-extent = #'(0 . 0)
    \override TimeSignature #'X-offset = #ly:self-alignment-interface::x-aligned-on-self
    \override TimeSignature #'Y-extent = #'(0 . 0)
    \override TimeSignature #'break-align-symbol = ##f
    \override TimeSignature #'break-visibility = #end-of-line-invisible
    \override TimeSignature #'font-size = #1
    \override TimeSignature #'self-alignment-X = #center
    \override VerticalAxisGroup #'default-staff-staff-spacing = #'((basic-distance . 0) (minimum-dis
  )
  \context {
    \Score
    \remove Bar_number_engraver
    \accepts TimeSignatureContext
    \override Beam #'breakable = ##t
    \override SpacingSpanner #'strict-grace-spacing = ##t
    \override SpacingSpanner #'strict-note-spacing = ##t
    \override SpacingSpanner #'uniform-stretching = ##t
    \override TupletBracket #'bracket-visibility = ##t
    \override TupletBracket #'minimum-length = #3
    \override TupletBracket #'padding = #2
    \override TupletBracket #'springs-and-rods = #ly:spanner::set-spacing-rods
    \override TupletNumber #'text = #tuplet-number::calc-fraction-text
    autoBeaming = ##f
    proportionalNotationDuration = #(ly:make-moment 1 32)
    tupletFullLength = ##t
  }
  \context {
    \StaffGroup
  }
  \context {
    \Staff
    \remove Time_signature_engraver
  }
  \context {
    \RhythmicStaff
    \remove Time_signature_engraver
  }
}

\paper {
  left-margin = #20
  system-system-spacing = #'((basic-distance . 0) (minimum-distance . 0) (padding . 12) (stretchability
}

\score {
  \new Score <<
    \new TimeSignatureContext {
      {
        \time 2/8
        s1 * 1/4
      }
      {
        \time 3/8

```

```

        s1 * 3/8
    }
    {
        \time 4/8
        s1 * 1/2
    }
}
\new Staff {
    {
        \time 2/8
        c'8 (
        d'8 )
    }
    {
        \time 3/8
        e'8 (
        f'8
        g'8 )
    }
    {
        \time 4/8
        fs'4 (
        e'8
        d'8 )
    }
}
>>
}

```

```
>>> show(lilypond_file)
```



Makes LilyPond file.

Wraps *music* in LilyPond `\score` block.

Adds LilyPond `\header`, `\layout`, `\paper` and `\score` blocks to LilyPond file.

Defines layout settings for custom `\TimeSignatureContext`.

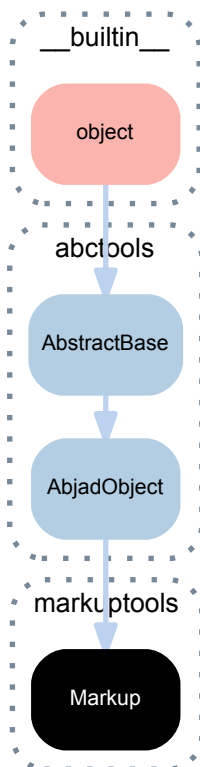
(Note that you must create and populate an Abjad context with name equal to `'TimeSignatureContext'` in order for `\TimeSignatureContext` layout settings to apply.)

Applies many file, layout and paper settings.

Returns LilyPond file.

9.1 Concrete classes

9.1.1 markuptools.Markup



class markuptools.**Markup** (*contents=None, direction=None*)
A LilyPond markup.

Initializes from string:

```
>>> string = r'\bold { "This is markup text." }'
>>> markup = Markup(string)
>>> show(markup)
```

This is markup text.

Initializes from other markup:

```
>>> markup_1 = Markup('foo', direction=Up)
>>> markup_2 = Markup(markup_1, direction=Down)
>>> show(markup_2)
```

foo

Attaches markup to score components:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> string = r'\italic { "This is also markup text." }'
>>> markup = Markup(string, direction=Up)
>>> attach(markup, staff[0])
>>> show(staff)
```



Set *direction* to Up, Down, 'neutral', '^', '_', '-' or None.

Markup objects are immutable.

Bases

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

Read-only properties

`Markup.contents`

Gets contents of markup.

```
>>> string = r'\bold { "This is markup text." }'
>>> markup = Markup(string)
>>> show(markup)
```

This is markup text.

```
>>> markup.contents
(MarkupCommand('bold', ['This is markup text.']),)
```

Returns tuple.

`Markup.direction`

Gets direction of markup.

```
>>> string = r'\bold { "This is markup text." }'
>>> markup = Markup(string, direction=Up)
>>> show(markup)
```

This is markup text.

```
>>> markup.direction
Up
```

Returns up, down, center or none.

Special methods

`Markup.__copy__ (*args)`

Copies markup.

```
>>> import copy
>>> string = r'\bold { allegro ma non troppo }'
>>> markup = Markup(string)
>>> new_markup = copy.copy(markup)
>>> show(new_markup)
```


allegro ma non troppo

Returns new markup.

Markup.__eq__(*expr*)

Is true when *expr* is a markup with format equal to that of this markup. Otherwise false.

Returns boolean.

Markup.__format__(*format_specification*='')

Formats markup.

```
>>> string = r'\bold { allegro ma non troppo }'
>>> markup = Markup(string)
>>> print(format(markup))
\markup {
  \bold
  {
    allegro
    ma
    non
    troppo
  }
}
```

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

Returns string.

Markup.__hash__()

Hashes markup.

Returns integer.

Markup.__illustrate__()

Illustrates markup.

```
>>> string = r'\bold { allegro ma non troppo }'
>>> markup = Markup(string)
>>> show(markup)
```

allegro ma non troppo

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.

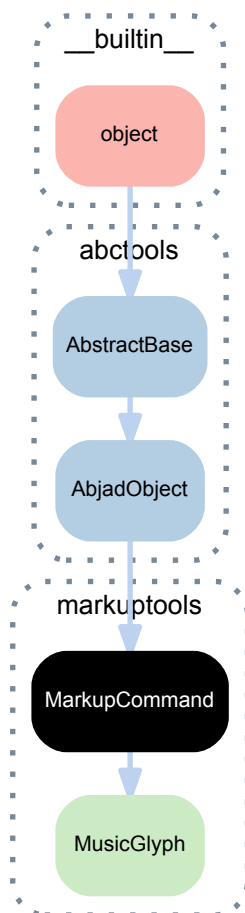
Markup.__str__()

Gets string representation of markup.

```
>>> string = r'\bold { allegro ma non troppo }'
>>> markup = Markup(string)
>>> print(str(markup))
\markup {
  \bold
  {
    allegro
    ma
    non
    troppo
  }
}
```

Returns string.

9.1.2 markuptools.MarkupCommand



class `markuptools.MarkupCommand` (*command=None, *args*)

A LilyPond markup command.

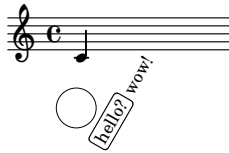
This creates a LilyPond markup command:

```
>>> 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`MarkupCommand.args`

Markup command arguments.

Returns tuple.

`MarkupCommand.command`

Markup command name.

Returns string.

Read/write properties

`MarkupCommand.force_quotes`

Is true when markup command should force quotes around arguments. Otherwise false.

Here's a markup command formatted in the usual way without forced quotes:

```
>>> lines = ['foo', 'bar blah', 'baz']
>>> command = markuptools.MarkupCommand('column', lines)
>>> markup = Markup(command)
```

```
>>> f(markup)
\markup {
  \column
  {
    foo
    "bar blah"
    baz
  }
}
```

The markup command forces quotes around only the spaced string 'bar blah'.

Here's the same markup command with forced quotes:

```
>>> lines = ['foo', 'bar blah', 'baz']
>>> command = markuptools.MarkupCommand('column', lines)
>>> command.force_quotes = True
>>> markup = Markup(command)
```

```
>>> f(markup)
\markup {
  \column
  {
    "foo"
    "bar blah"
    "baz"
  }
}
```

```
}
}
```

The markup command forces quotes around all strings.

The rendered result of forced and unforced quotes is the same.

Defaults to false.

Returns boolean.

Special methods

`MarkupCommand.__eq__(expr)`

Is 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.

`MarkupCommand.__hash__()`

Hashes markup command.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

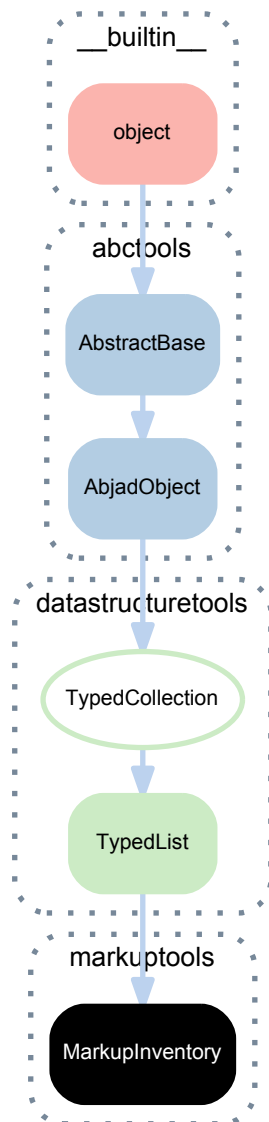
Returns string.

`MarkupCommand.__str__()`

Gets string representation of markup command.

Returns string.

9.1.3 markuptools.MarkupInventory



class markuptools.**MarkupInventory** (*items=None, item_class=None, keep_sorted=None*)
 Abjad model of an ordered list of markup:

```
>>> inventory = markuptools.MarkupInventory(['foo', 'bar'])
```

```
>>> inventory
MarkupInventory([Markup(contents='foo',), Markup(contents='bar',)])
```

Markup inventories implement the list interface and are mutable.

Bases

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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(item)`
Changes *item* to item and returns index.

```
>>> pitch_collection = datastructuretools.TypedList(  
...     items=('cqf', "as'", 'b', 'dss'),  
...     item_class=NamedPitch)  
>>> pitch_collection.index("as'")  
1
```

Returns index.

(TypedList) **.insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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__** (*item*)

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

Returns none.

(TypedCollection) **.__eq__** (*expr*)

Is 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__**().

Returns item.

(TypedCollection).**__hash__**()
 Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedList).**__iadd__**(*expr*)
 Changes items 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(
            name='ppp',
        ),
        indicatortools.Dynamic(
            name='p',
        ),
        indicatortools.Dynamic(
            name='mp',
        ),
        indicatortools.Dynamic(
            name='mf',
        ),
        indicatortools.Dynamic(
            name='fff',
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

MarkupInventory.**__illustrate__**()
 Illustrates markup inventory.

```
>>> inventory = markuptools.MarkupInventory(['foo', 'bar'])
>>> show(inventory)
```

```
foo
bar
```

Returns LilyPond file.

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

Returns generator.

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

Returns nonnegative integer.

(TypedCollection).**__ne__**(*expr*)
 Is 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__**().

Returns generator.

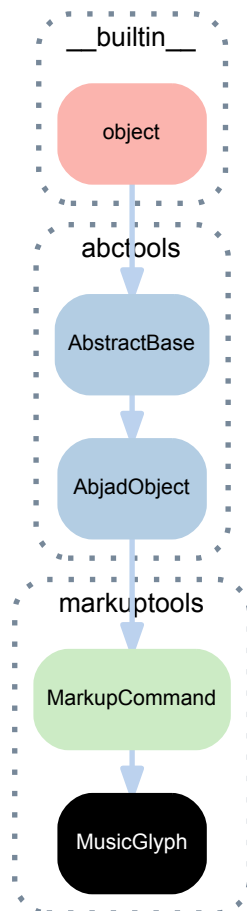
(TypedList).**__setitem__**(*i, expr*)
 Changes items in *expr* to items and sets.

```
>>> 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,
)
```

Returns none.

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
- abctools.AbjadObject.AbstractBase
- __builtin__.object

Read-only properties

(MarkupCommand) **.args**
 Markup command arguments.
 Returns tuple.

(MarkupCommand) **.command**
 Markup command name.
 Returns string.

Read/write properties

(MarkupCommand) .**force_quotes**

Is true when markup command should force quotes around arguments. Otherwise false.

Here's a markup command formatted in the usual way without forced quotes:

```
>>> lines = ['foo', 'bar blah', 'baz']
>>> command = markuptools.MarkupCommand('column', lines)
>>> markup = Markup(command)
```

```
>>> f(markup)
\markup {
  \column
  {
    foo
    "bar blah"
    baz
  }
}
```

The markup command forces quotes around only the spaced string 'bar blah'.

Here's the same markup command with forced quotes:

```
>>> lines = ['foo', 'bar blah', 'baz']
>>> command = markuptools.MarkupCommand('column', lines)
>>> command.force_quotes = True
>>> markup = Markup(command)
```

```
>>> f(markup)
\markup {
  \column
  {
    "foo"
    "bar blah"
    "baz"
  }
}
```

The markup command forces quotes around all strings.

The rendered result of forced and unforced quotes is the same.

Defaults to false.

Returns boolean.

Special methods

(MarkupCommand) .**__eq__**(*expr*)

Is 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.

(MarkupCommand) .**__hash__**()

Hashes markup command.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

(MarkupCommand) .**__str__**()
Gets 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 \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(contents=(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)`

Makes vertically adjusted *composer* markup.

```
>>> markup = markuptools.make_vertically_adjusted_composer_markup('Josquin Desprez')
```

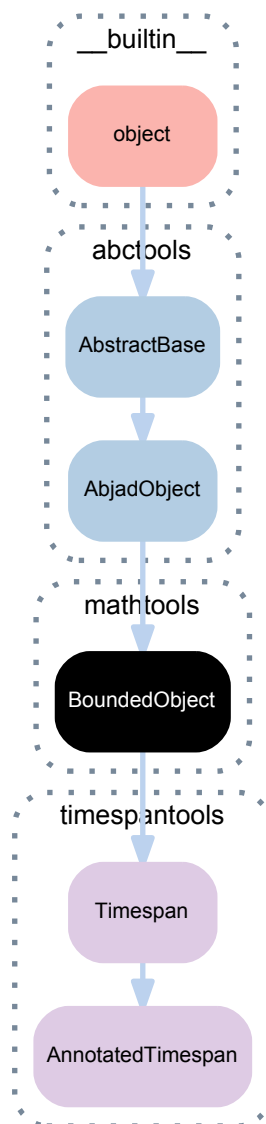
```
>>> print(format(markup, 'lilypond'))
\markup {
  \override
    #'(font-name . "Times")
    {
```

```
        \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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`BoundedObject.is_closed`

Is true when left closed and right closed. Otherwise false.

Returns boolean.

`BoundedObject.is_half_closed`

Is true when left closed xor right closed.

Returns boolean.

`BoundedObject.is_half_open`

Is true when left and right open are not the same. Otherwise false.

Return boolean.

`BoundedObject.is_open`

Is true when left or right open. Otherwise false.

Returns boolean.

Read/write properties

`BoundedObject.is_left_closed`

Is true when left closed. Otherwise false.

Returns boolean.

`BoundedObject.is_left_open`

Is true when left open. Otherwise false.

Returns boolean.

`BoundedObject.is_right_closed`

Is true when right closed. Otherwise false.

Returns boolean.

`BoundedObject.is_right_open`

Is 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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

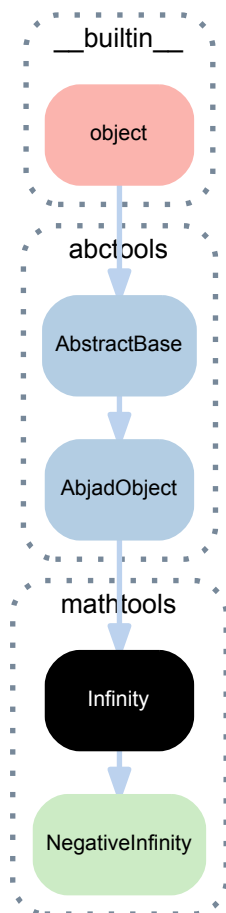
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

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

Special methods

`Infinity.__eq__(expr)`

Is true when *expr* is also infinity. Otherwise false.

Returns boolean.

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

Formats Abjad 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.__hash__()`

Hashes infinity.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`Infinity.__le__(expr)`

Is 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.

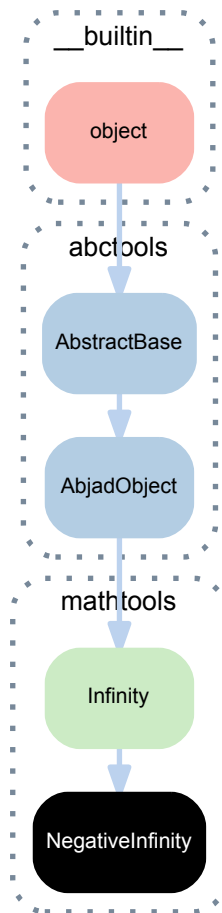
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

`Infinity.__sub__(expr)`
 Subtracts *expr* from infinity.
 Returns infinity or 0 if *expr* is also infinity.

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`

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

Special methods

`(Infinity).__eq__(expr)`

Is true when *expr* is also infinity. Otherwise false.

Returns boolean.

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

Formats Abjad 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).__hash__()`

Hashes infinity.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(Infinity).__le__(expr)`

Is 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.

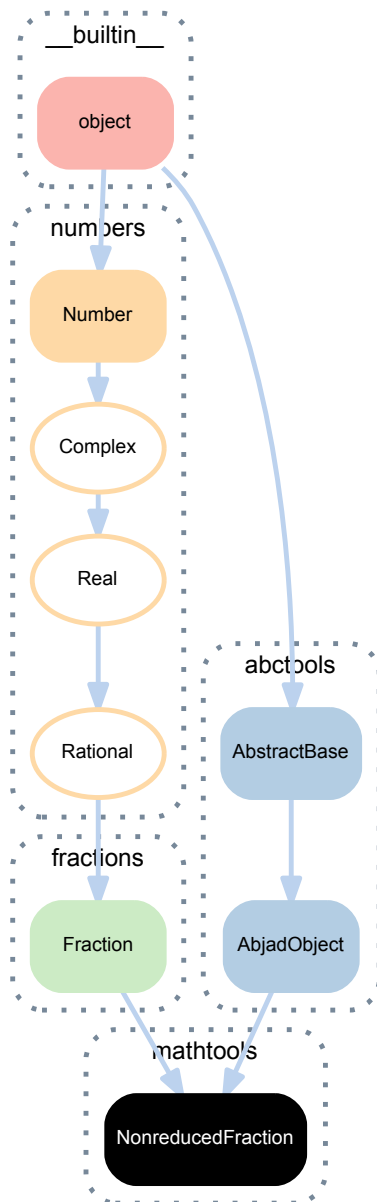
Returns string.

`(Infinity).__sub__(expr)`

Subtracts *expr* from infinity.

Returns infinity or 0 if *expr* is also infinity.

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
```

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `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)
```

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)`
Is 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)`
Is true when nonreduced fraction is greater than or equal to *expr*.

```
>>> mathtools.NonreducedFraction(3, 3) >= 1
True
```

Returns boolean.

`NonreducedFraction.__gt__(expr)`
Is true when nonreduced fraction is greater than *expr*.

```
>>> mathtools.NonreducedFraction(3, 3) > 1
False
```

Returns boolean.

`NonreducedFraction.__hash__()`
Hashes nonreduced fraction.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`NonreducedFraction.__le__(expr)`
Is true when nonreduced fraction is less than or equal to *expr*.

```
>>> mathtools.NonreducedFraction(3, 3) <= 1
True
```

Returns boolean.

`NonreducedFraction.__lt__(expr)`
Is 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)`
Is 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.

`NonreducedFraction.__rtruediv__(expr)`

Divides *expr* by nonreduced fraction.

```
>>> 1 / mathtools.NonreducedFraction(3, 3)
NonreducedFraction(3, 3)
```

Returns nonreduced fraction.

`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)
```

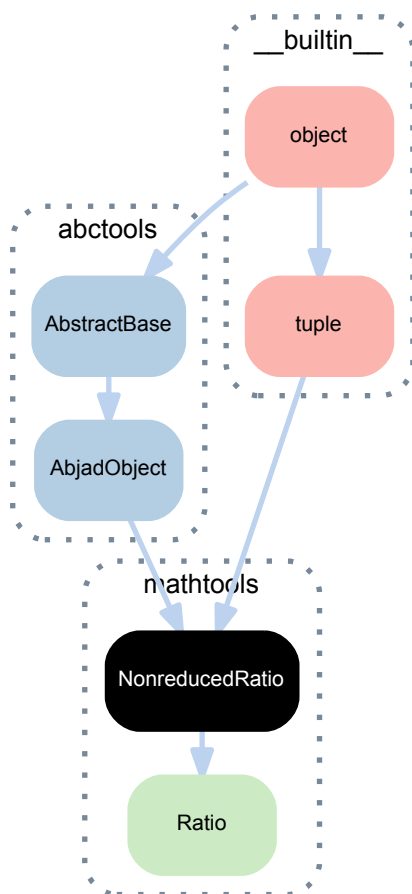
Returns nonreduced fraction.

`NonreducedFraction.__truediv__(expr)`
Divides nonreduced fraction in Python 3.

Returns nonreduced fraction.

`(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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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
```

```
NonreducedRatio.__hash__()
Hashes non-reduced ratio.
```

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

```
(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
```

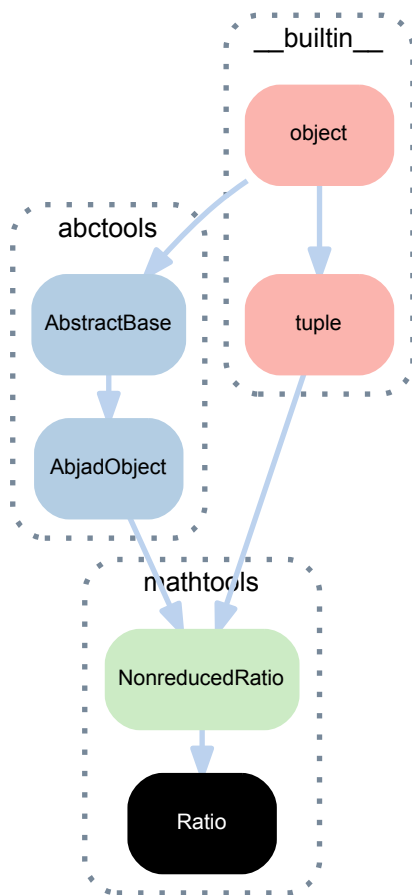
```
(AbjadObject) .__ne__(expr)
Is true when Abjad object does not equal expr. Otherwise false.
Returns boolean.
```

```
NonreducedRatio.__new__(*args)
```

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

```
(tuple) .__rmul__()
x.__rmul__(n) <==> 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)
Is 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

(NonreducedRatio).**__hash__**()
Hashes non-reduced ratio.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(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__**()
 $x._\text{rmul}_\text{(n)} \iff n * x$

Ratio.**__str__**()
 String representation of ratio.
 Returns string.

10.2 Functions

10.2.1 `mathtools.all_are_equal`

`mathtools.all_are_equal(expr)`
 Is true when *expr* is a sequence and all elements in *expr* are equal:

```
>>> mathtools.all_are_equal([99, 99, 99, 99, 99, 99])
True
```

Is true when *expr* is an empty sequence:

```
>>> mathtools.all_are_equal([])
True
```

Otherwise false:

```
>>> mathtools.all_are_equal(17)
False
```

Returns boolean.

10.2.2 `mathtools.all_are_integer_equivalent_exprs`

`mathtools.all_are_integer_equivalent_exprs(expr)`
 Is true when *expr* is a sequence and all elements in *expr* are integer-equivalent expressions.

```
>>> mathtools.all_are_integer_equivalent_exprs([1, '2', 3.0, Fraction(4, 1)])
True
```

Otherwise false:

```
>>> mathtools.all_are_integer_equivalent_exprs([1, '2', 3.5, 4])
False
```

Returns boolean.

10.2.3 `mathtools.all_are_integer_equivalent_numbers`

`mathtools.all_are_integer_equivalent_numbers(expr)`
 Is true when *expr* is a sequence and all elements in *expr* are integer-equivalent numbers.

```
>>> mathtools.all_are_integer_equivalent_numbers([1, 2, 3.0, Fraction(4, 1)])
True
```

Otherwise false:

```
>>> mathtools.all_are_integer_equivalent_numbers([1, 2, 3.5, 4])
False
```

Returns boolean.

10.2.4 `mathtools.all_are_nonnegative_integer_equivalent_numbers`

`mathtools.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)]
>>> mathtools.all_are_nonnegative_integer_equivalent_numbers(expr)
True
```

Returns boolean.

10.2.5 `mathtools.all_are_nonnegative_integer_powers_of_two`

`mathtools.all_are_nonnegative_integer_powers_of_two` (*expr*)

Is true when *expr* is a sequence and all elements in *expr* are nonnegative integer powers of two.

```
>>> mathtools.all_are_nonnegative_integer_powers_of_two([0, 1, 1, 1, 2, 4, 32, 32])
True
```

Is true when *expr* is an empty sequence:

```
>>> mathtools.all_are_nonnegative_integer_powers_of_two([])
True
```

Otherwise false:

```
>>> mathtools.all_are_nonnegative_integer_powers_of_two(17)
False
```

Returns boolean.

10.2.6 `mathtools.all_are_nonnegative_integers`

`mathtools.all_are_nonnegative_integers` (*expr*)

Is true when *expr* is a sequence and all elements in *expr* are nonnegative integers.

```
>>> mathtools.all_are_nonnegative_integers([0, 1, 2, 99])
True
```

Otherwise false:

```
>>> mathtools.all_are_nonnegative_integers([0, 1, 2, -99])
False
```

Returns boolean.

10.2.7 `mathtools.all_are_numbers`

`mathtools.all_are_numbers` (*expr*)

Is true when *expr* is a sequence and all elements in *expr* are numbers:

```
>>> mathtools.all_are_numbers([1, 2, 3.0, Fraction(13, 8)])
True
```

Is true when *expr* is an empty sequence:

```
>>> mathtools.all_are_numbers([])
True
```

Otherwise false:

```
>>> mathtools.all_are_numbers(17)
False
```

Returns boolean.

10.2.8 `mathtools.all_are_pairs`

`mathtools.all_are_pairs` (*expr*)

Is true when *expr* is a sequence whose members are all sequences of length 2.

```
>>> mathtools.all_are_pairs([(1, 2), (3, 4), (5, 6), (7, 8)])
True
```

Is true when *expr* is an empty sequence:

```
>>> mathtools.all_are_pairs([])
True
```

Otherwise false:

```
>>> mathtools.all_are_pairs('foo')
False
```

Returns boolean.

10.2.9 `mathtools.all_are_pairs_of_types`

`mathtools.all_are_pairs_of_types` (*expr*, *first_type*, *second_type*)

Is 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*.

```
>>> mathtools.all_are_pairs_of_types([(1., 'a'), (2.1, 'b'), (3.45, 'c')], float, str)
True
```

Is true when *expr* is an empty sequence:

```
>>> mathtools.all_are_pairs_of_types([], float, str)
True
```

Otherwise false:

```
>>> mathtools.all_are_pairs_of_types('foo', float, str)
False
```

Returns boolean.

10.2.10 `mathtools.all_are_positive_integer_equivalent_numbers`

`mathtools.all_are_positive_integer_equivalent_numbers` (*expr*)

Is true when *expr* is a sequence and all elements in *expr* are positive integer-equivalent numbers. Otherwise false:

```
>>> mathtools.all_are_positive_integer_equivalent_numbers([Fraction(4, 2), 2.0, 2])
True
```

Returns boolean.

10.2.11 `mathtools.all_are_positive_integer_powers_of_two`

`mathtools.all_are_positive_integer_powers_of_two` (*expr*)

Is true when *expr* is a sequence and all elements in *expr* are positive integer powers of two.

```
>>> mathtools.all_are_nonnegative_integer_powers_of_two([1, 1, 1, 2, 4, 32, 32])
True
```

Is true when *expr* is an empty sequence:

```
>>> mathtools.all_are_nonnegative_integer_powers_of_two([])
True
```

Otherwise false:

```
>>> mathtools.all_are_nonnegative_integer_powers_of_two(17)
False
```

Returns boolean.

10.2.12 `mathtools.all_are_positive_integers`

`mathtools.all_are_positive_integers` (*expr*)

Is true when *expr* is a sequence and all elements in *expr* are positive integers.

```
>>> mathtools.all_are_positive_integers([1, 2, 3, 99])
True
```

Otherwise false:

```
>>> mathtools.all_are_positive_integers(17)
False
```

Returns boolean.

10.2.13 `mathtools.all_are_unequal`

`mathtools.all_are_unequal` (*expr*)

Is true when *expr* is a sequence all elements in *expr* are unequal.

```
>>> mathtools.all_are_unequal([1, 2, 3, 4, 9])
True
```

Is true when *expr* is an empty sequence:

```
>>> mathtools.all_are_unequal([])
True
```

Otherwise false:

```
>>> mathtools.all_are_unequal(17)
False
```

Returns boolean.

10.2.14 `mathtools.are_relatively_prime`

`mathtools.are_relatively_prime` (*expr*)

Is 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.

10.2.15 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.

10.2.16 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.

10.2.17 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.

10.2.18 `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.

10.2.19 `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.

10.2.20 `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.

10.2.21 `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.

10.2.22 `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.

10.2.23 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]
```

Raises type error on noninteger *n*.

Raises not implemented error on 0.

Returns list of positive integers.

10.2.24 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.

10.2.25 `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.

10.2.26 `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.

10.2.27 `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.

10.2.28 `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.

10.2.29 `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.

10.2.30 `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.

10.2.31 `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.

10.2.32 `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.

10.2.33 `mathtools.is_assignable_integer`

`mathtools.is_assignable_integer` (*expr*)
Is 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.

10.2.34 `mathtools.is_dotted_integer`

`mathtools.is_dotted_integer(expr)`

Is 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 $0 \leq j$, $2 < k$.

10.2.35 `mathtools.is_fraction_equivalent_pair`

`mathtools.is_fraction_equivalent_pair(expr)`

Is true when *expr* is an integer-equivalent pair of numbers excluding 0 as the second term.

```
>>> mathtools.is_fraction_equivalent_pair((2, 3))
True
```

Otherwise false:

```
>>> mathtools.is_fraction_equivalent_pair((2, 0))
False
```

Returns boolean.

10.2.36 `mathtools.is_integer_equivalent_expr`

`mathtools.is_integer_equivalent_expr(expr)`
Is true when *expr* is an integer-equivalent number.

```
>>> mathtools.is_integer_equivalent_expr(12.0)
True
```

Is 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.

10.2.37 `mathtools.is_integer_equivalent_n_tuple`

`mathtools.is_integer_equivalent_n_tuple(expr, n)`
Is true when *expr* is a tuple of *n* integer-equivalent expressions.

```
>>> mathtools.is_integer_equivalent_n_tuple((2.0, '3', Fraction(4, 1)), 3)
True
```

Otherwise false:

```
>>> mathtools.is_integer_equivalent_n_tuple((2.5, '3', Fraction(4, 1)), 3)
False
```

Returns boolean.

10.2.38 `mathtools.is_integer_equivalent_number`

`mathtools.is_integer_equivalent_number(expr)`
Is 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.

10.2.39 `mathtools.is_integer_equivalent_pair`

`mathtools.is_integer_equivalent_pair(expr)`
Is true when *expr* is a pair of integer-equivalent expressions.

```
>>> mathtools.is_integer_equivalent_pair((2.0, '3'))
True
```

Otherwise false:

```
>>> mathtools.is_integer_equivalent_pair((2.5, '3'))
False
```

Returns boolean.

10.2.40 `mathtools.is_integer_equivalent_singleton`

`mathtools.is_integer_equivalent_singleton(expr)`

Is true when *expr* is a singleton of integer-equivalent expressions.

```
>>> mathtools.is_integer_equivalent_singleton((2.0,))
True
```

Otherwise false:

```
>>> mathtools.is_integer_equivalent_singleton((2.5,))
False
```

Returns boolean.

10.2.41 `mathtools.is_integer_n_tuple`

`mathtools.is_integer_n_tuple(expr, n)`

Is true when *expr* is an integer tuple of length *n*.

```
>>> mathtools.is_integer_n_tuple((19, 20, 21), 3)
True
```

Otherwise false:

```
>>> mathtools.is_integer_n_tuple((19, 20, 'text'), 3)
False
```

Returns boolean.

10.2.42 `mathtools.is_integer_pair`

`mathtools.is_integer_pair(expr)`

Is true when *expr* is an integer tuple of length 2.

```
>>> mathtools.is_integer_pair((19, 20))
True
```

Otherwise false:

```
>>> mathtools.is_integer_pair(('some', 'text'))
False
```

Returns boolean.

10.2.43 `mathtools.is_integer_singleton`

`mathtools.is_integer_singleton(expr)`

Is true when *expr* is an integer tuple of of length 1.

```
>>> mathtools.is_integer_singleton((19,))
True
```

Otherwise false:

```
>>> mathtools.is_integer_singleton(('text',))
False
```

Returns boolean.

10.2.44 `mathtools.is_n_tuple`

`mathtools.is_n_tuple(expr, n)`

Is true when *expr* is a tuple of length *n*.

```
>>> mathtools.is_n_tuple((19, 20, 21), 3)
True
```

Otherwise false:

```
>>> mathtools.is_n_tuple((19, 20, 21), 4)
False
```

Returns boolean.

10.2.45 `mathtools.is_negative_integer`

`mathtools.is_negative_integer(expr)`

Is 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.

10.2.46 `mathtools.is_nonnegative_integer`

`mathtools.is_nonnegative_integer(expr)`

Is 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.

10.2.47 `mathtools.is_nonnegative_integer_equivalent_number`

`mathtools.is_nonnegative_integer_equivalent_number(expr)`

Is true when *expr* is a nonnegative integer-equivalent number. Otherwise false:

```
>>> mathtools.is_nonnegative_integer_equivalent_number(Duration(4, 2))
True
```

Returns boolean.

10.2.48 `mathtools.is_nonnegative_integer_power_of_two`

`mathtools.is_nonnegative_integer_power_of_two(expr)`

Is 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.

10.2.49 `mathtools.is_null_tuple`

`mathtools.is_null_tuple(expr)`

Is true when *expr* is a tuple of length 0.

```
>>> mathtools.is_null_tuple(())
True
```

Otherwise false:

```
>>> mathtools.is_null_tuple((19, 20, 21))
False
```

Returns boolean.

10.2.50 `mathtools.is_pair`

`mathtools.is_pair(expr)`

Is true when *expr* is a tuple of length 2.

```
>>> mathtools.is_pair((19, 20))
True
```

Otherwise false:

```
>>> mathtools.is_pair((19, 20, 21))
False
```

Returns boolean.

10.2.51 `mathtools.is_positive_integer`

`mathtools.is_positive_integer(expr)`

Is 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.

10.2.52 `mathtools.is_positive_integer_equivalent_number`

`mathtools.is_positive_integer_equivalent_number` (*expr*)
Is true when *expr* is a positive integer-equivalent number. Otherwise false:

```
>>> mathtools.is_positive_integer_equivalent_number(Duration(4, 2))
True
```

Returns boolean.

10.2.53 `mathtools.is_positive_integer_power_of_two`

`mathtools.is_positive_integer_power_of_two` (*expr*)
Is 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.

10.2.54 `mathtools.is_singleton`

`mathtools.is_singleton` (*expr*)
Is true when *expr* is a tuple of length 1.

```
>>> mathtools.is_singleton((19,))
True
```

Otherwise false:

```
>>> mathtools.is_singleton((19, 20, 21))
False
```

Returns boolean.

10.2.55 `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.

10.2.56 `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.

10.2.57 `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.

10.2.58 `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.

10.2.59 `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.

10.2.60 `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.

10.2.61 `mathtools.partition_integer_into_halves`

`mathtools.partition_integer_into_halves` (*n*, *bigger*='left', *even*='allowed')

Writes positive integer *n* as the pair *t* = (*left*, *right*) such that *n* == *left* + *right*.

When *n* is odd the greater part of *t* corresponds to the value of *bigger*:

```
>>> mathtools.partition_integer_into_halves(7, bigger='left')
(4, 3)
>>> mathtools.partition_integer_into_halves(7, bigger='right')
(3, 4)
```

Likewise when *n* is even and *even* = 'disallowed':

```
>>> mathtools.partition_integer_into_halves(8, bigger='left', even='disallowed')
(5, 3)
>>> mathtools.partition_integer_into_halves(8, bigger='right', even='disallowed')
(3, 5)
```

But when *n* is even and *even* = 'allowed' then *left* == *right* and *bigger* is ignored:

```
>>> mathtools.partition_integer_into_halves(8)
(4, 4)
>>> mathtools.partition_integer_into_halves(8, bigger='left')
(4, 4)
>>> mathtools.partition_integer_into_halves(8, bigger='right')
(4, 4)
```

When *n* is 0 return (0, 0):

```
>>> mathtools.partition_integer_into_halves(0)
(0, 0)
```

When *n* is 0 and *even* = 'disallowed' raises partition error.

Raises type error on noninteger *n*.

Raises value error on negative *n*.

Returns pair of positive integers.

10.2.62 `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.

10.2.63 `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.

10.2.64 `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.

10.2.65 `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 .

10.2.66 `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.

10.2.67 `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.

10.2.68 `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.

10.2.69 `mathtools.yield_nonreduced_fractions`

`mathtools.yield_nonreduced_fractions()`

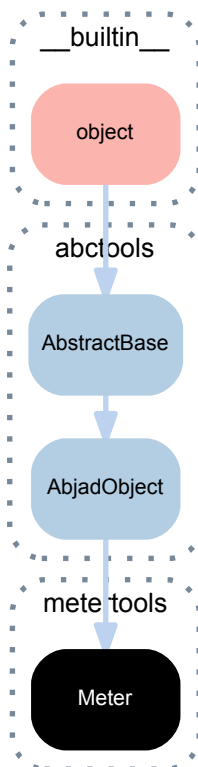
Yields positive nonreduced fractions in Cantor diagonalized order:

```
>>> generator = mathtools.yield_nonreduced_fractions()
>>> for n in range(16):
...     next(generator)
...
(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((7, 4))
>>> print(meter.pretty_rtm_format)
(7/4 (
  (3/4 (
    1/4
    1/4
    1/4))
  (2/4 (
    1/4
    1/4))
  (2/4 (
    1/4
    1/4))))
```

```
>>> meter = metertools.Meter(
...   (7, 4), decrease_durations_monotonically=False)
>>> print(meter.pretty_rtm_format)
(7/4 (
  (2/4 (
    1/4
    1/4))
  (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`
- `abctools.AbjadObject.AbstractBase`
- `__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(
...     (7, 4),
...     decrease_durations_monotonically=False,
... )
```

```
>>> meter.decrease_durations_monotonically
False
```

```
>>> print(meter.pretty_rtm_format)
(7/4 (
  (2/4 (
    1/4
    1/4))
  (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((7, 4),
...     decrease_durations_monotonically=True)
```

```
>>> meter.decrease_durations_monotonically
True
```

```
>>> print(meter.pretty_rtm_format)
(7/4 (
  (3/4 (
    1/4
    1/4
    1/4))
  (2/4 (
    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(7, 4))
1 (Offset(0, 1), Offset(3, 4), Offset(5, 4), Offset(7, 4))
2 (Offset(0, 1), Offset(1, 4), Offset(1, 2), Offset(3, 4), Offset(1, 1), Offset(5, 4), Offset(3, 2), Off
```

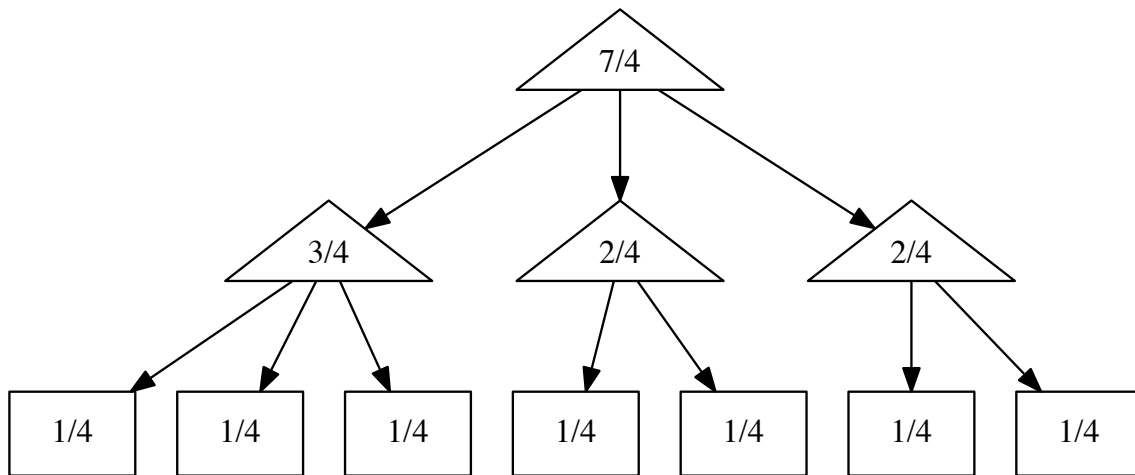
Returns dictionary.

Meter.graphviz_format

Graphviz format of hierarchy's root node:

```
>>> print(meter.graphviz_format)
digraph G {
    node_0 [label="7/4",
            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_8 [label="2/4",
            shape=triangle];
    node_9 [label="1/4",
            shape=box];
    node_10 [label="1/4",
             shape=box];
    node_0 -> node_1;
    node_0 -> node_5;
    node_0 -> node_8;
    node_1 -> node_2;
    node_1 -> node_3;
    node_1 -> node_4;
    node_5 -> node_6;
    node_5 -> node_7;
    node_8 -> node_10;
    node_8 -> node_9;
}
```

```
>>> 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
7
```

Returns positive integer.

Meter.preprolated_duration

Beat hierarchy preprolated_duration:

```
>>> meter.preprolated_duration
Duration(7, 4)
```

Returns preprolated_duration.

Meter.pretty_rtm_format

Beat hierarchy pretty RTM format:

```
>>> print(meter.pretty_rtm_format)
(7/4 (
  (3/4 (
    1/4
    1/4
    1/4))
  (2/4 (
    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)
),
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(7, 4)
)

```

Returns rhythm tree node.

`Meter.rtm_format`

Beat hierarchy RTM format:

```

>>> meter.rtm_format
'(7/4 ((3/4 (1/4 1/4 1/4)) (2/4 (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.items()):
...     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, denominator=32, discard_final_orphan_downbeat=True, maximum_repetitions=None, starting_offset=None)`

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.implied_time_signature)
...
4/4
4/4
4/4
4/4
```

Example 2. Matching a series of hypothetical 5/4 measures:

```
>>> expr = [(0, 4), (3, 4), (5, 4), (10, 4), (15, 4), (20, 4)]
>>> for x in metertools.Meter.fit_meters_to_expr(
...     expr, meters):
...     print(x.implied_time_signature)
...
3/4
3/4
4/4
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)`

Is 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((7, 4))
>>> print(format(meter))
metertools.Meter(
  '(7/4 ((3/4 (1/4 1/4 1/4)) (2/4 (1/4 1/4)) (2/4 (1/4 1/4))))'
```

Returns string.

Meter.__hash__()

Hashes meter.

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(3, 4), NonreducedFraction(4, 4))
(NonreducedFraction(4, 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.

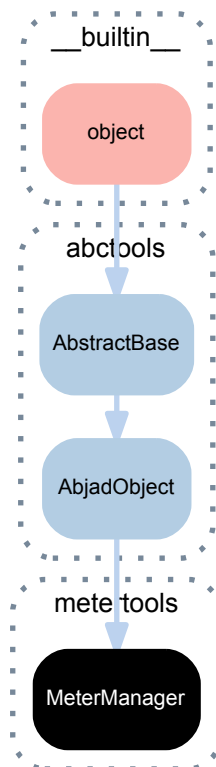
Returns boolean.

(AbjadObject).__repr__()

Gets interpreter representation of Abjad object.

Returns string.

11.1.2 metertools.MeterManager



class metertools.MeterManager
A meter manager.

Bases

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

Static methods

`MeterManager.is_acceptable_logical_tie` (*logical_tie_duration=None*, *logical_tie_starts_in_offsets=None*, *logical_tie_stops_in_offsets=None*, *maximum_dot_count=None*)

Is true if logical tie is acceptable.

`MeterManager.is_boundary_crossing_logical_tie` (*boundary_depth=None*, *boundary_offsets=None*, *logical_tie_start_offset=None*, *logical_tie_stop_offset=None*)

Is true if logical tie crosses meter boundaries.

`MeterManager.iterate_rewrite_inputs` (*expr*)

Iterate topmost masked logical ties, rest groups and containers in *expr*, masked by *expr*:

```

>>> string = "abj: | 2/4 c'4 d'4 ~ |"
>>> string += "| 4/4 d'8. r16 r8. e'16 ~ "
>>> string += "2/3 { e'8 ~ e'8 f'8 ~ } f'4 ~ |"
>>> string += "| 4/4 f'8 g'8 ~ g'4 a'4 ~ a'8 b'8 ~ |"
>>> string += "| 2/4 b'4 c'4 |"
>>> staff = Staff(string)
  
```

```
>>> for x in metertools.MeterManager.iterate_rewrite_inputs(
...     staff[0]): x
...
LogicalTie(Note("c'4"),)
LogicalTie(Note("d'4"),)
```

```
>>> for x in metertools.MeterManager.iterate_rewrite_inputs(
...     staff[1]): x
...
LogicalTie(Note("d'8."),)
LogicalTie(Rest('r16'), Rest('r8.'))
LogicalTie(Note("e'16"),)
Tuplet(Multiplier(2, 3), "e'8 ~ e'8 f'8 ~")
LogicalTie(Note("f'4"),)
```

```
>>> for x in metertools.MeterManager.iterate_rewrite_inputs(
...     staff[2]): x
...
LogicalTie(Note("f'8"),)
LogicalTie(Note("g'8"), Note("g'4"))
LogicalTie(Note("a'4"), Note("a'8"))
LogicalTie(Note("b'8"),)
```

```
>>> for x in metertools.MeterManager.iterate_rewrite_inputs(
...     staff[3]): x
...
LogicalTie(Note("b'4"),)
LogicalTie(Note("c'4"),)
```

Returns generator.

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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

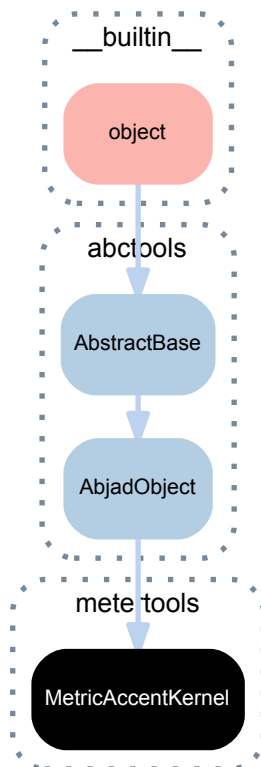
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

11.1.3 metertools.MetricAccentKernel



class metertools.**MetricAccentKernel** (*kernel=None*)

A metrical kernel, or offset-impulse-response-filter.

```

>>> hierarchy = metertools.Meter((7, 8))
>>> kernel = hierarchy.generate_offset_kernel_to_denominator(8)
>>> kernel
MetricAccentKernel(
  {
    Offset(0, 1): Multiplier(3, 14),
    Offset(1, 8): Multiplier(1, 14),
    Offset(1, 4): Multiplier(1, 14),
    Offset(3, 8): Multiplier(1, 7),
    Offset(1, 2): Multiplier(1, 14),
    Offset(5, 8): Multiplier(1, 7),
    Offset(3, 4): Multiplier(1, 14),
    Offset(7, 8): Multiplier(3, 14),
  }
)
  
```

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)
0.5
  
```

Return *MetricAccentKernel* instance.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

MetricAccentKernel.**__hash__**()

Hashes metric accent kernel.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

Returns boolean.

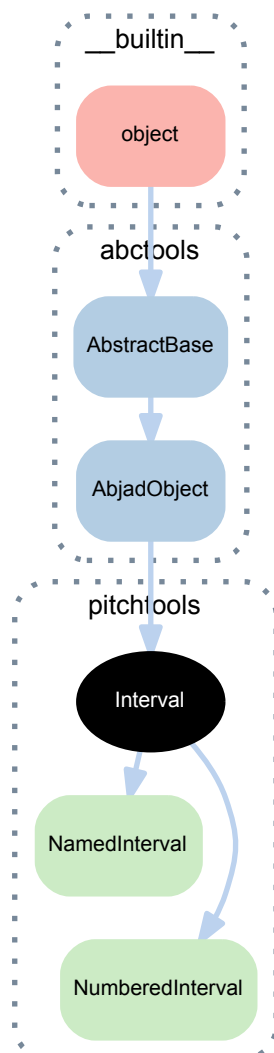
(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

12.1 Abstract classes

12.1.1 `pitchtools.Interval`



`class pitchtools.Interval`
Interval base class.

Bases

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

Read-only properties

`Interval.cents`

Cents of interval.

Returns nonnegative number.

Static methods

`Interval.is_named_interval_abbreviation(expr)`

Is 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)`

Is 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)`

Is 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 Abjad 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.

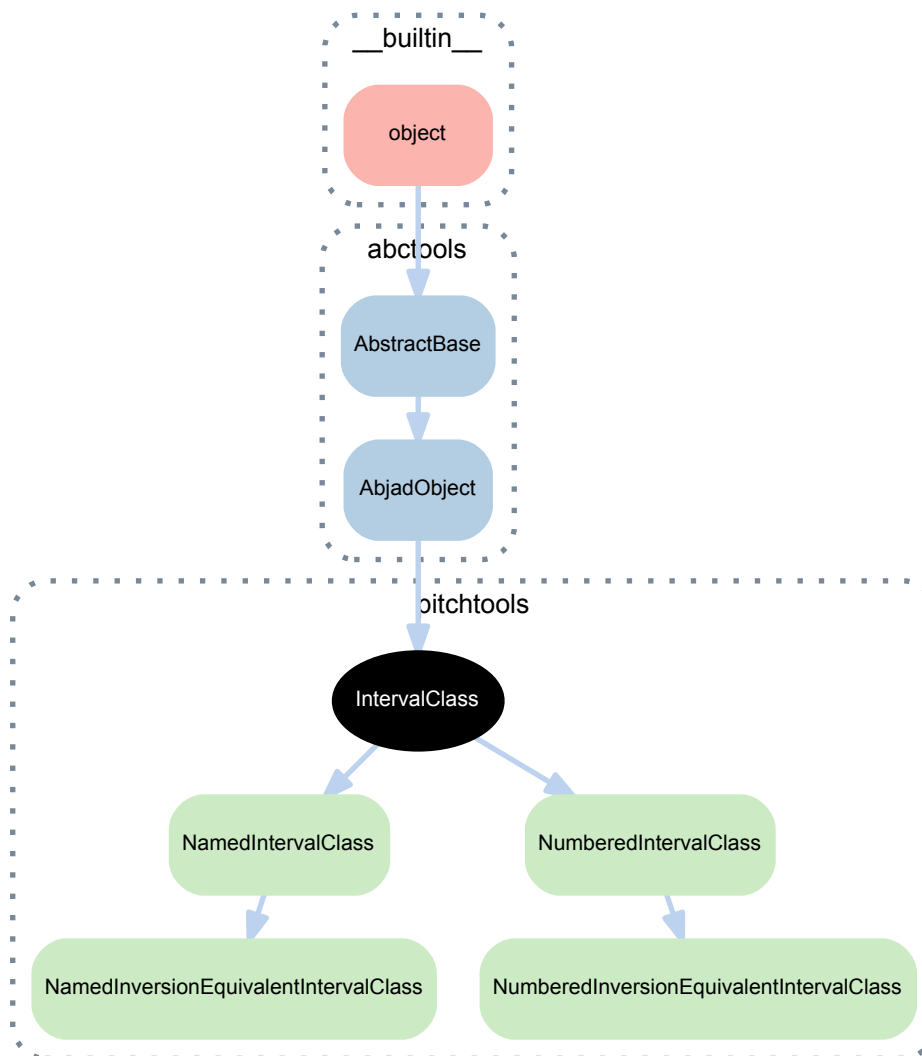
`Interval.__ne__(arg)`
 Is true when interval does not equal *arg*.
 Returns boolean.

`Interval.__neg__()`
 Negates interval.
 Returns interval.

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

`Interval.__str__()`
 String representation of interval.
 Returns string.

12.1.2 pitchtools.IntervalClass



class `pitchtools.IntervalClass`
Interval-class base class.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

`IntervalClass.__hash__()`
Hashes interval-class.
Returns integer.

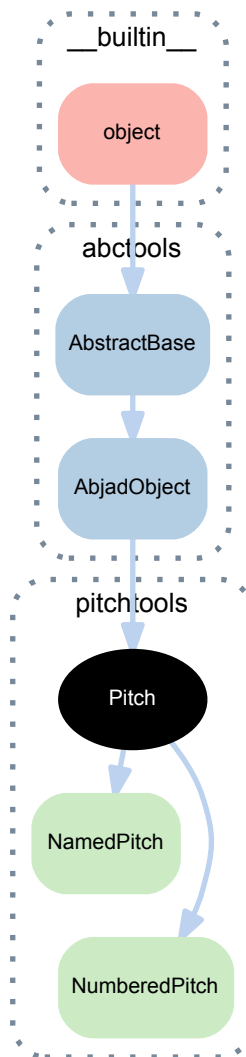
`IntervalClass.__int__()`
Change interval-class to integer.
Returns integer.

`(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.

12.1.3 pitchtools.Pitch



class `pitchtools.Pitch`
Pitch base class.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

Class methods

`Pitch.from_hertz` (*hertz*)

Creates pitch from *hertz*.

```
>>> pitchtools.NamedPitch.from_hertz(440)
NamedPitch("a'")
```

```
>>> pitchtools.NumberedPitch.from_hertz(440)
NumberedPitch(9)
```

Returns new pitch.

Static methods

`Pitch.is_diatonic_pitch_name` (*expr*)

Is 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*)

Is 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*)

Is 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*)

Is 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)`

Is 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

`(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.

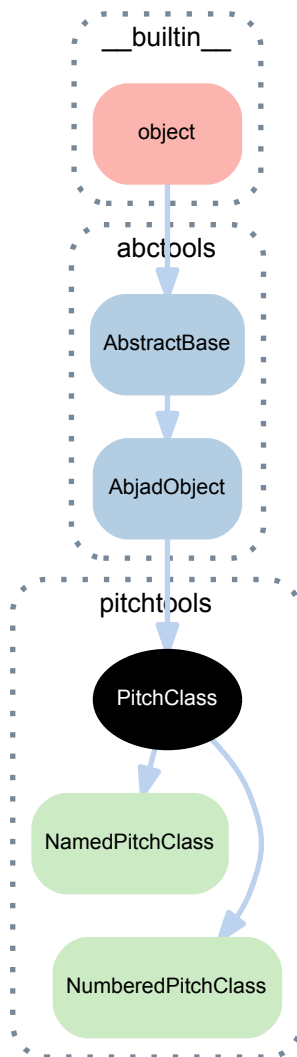
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

12.1.4 pitchtools.PitchClass



class `pitchtools.PitchClass`
Pitch-class base class.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)`
 Is 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)`
 Is 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)`

Is 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__()`

Hases pitch-class.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

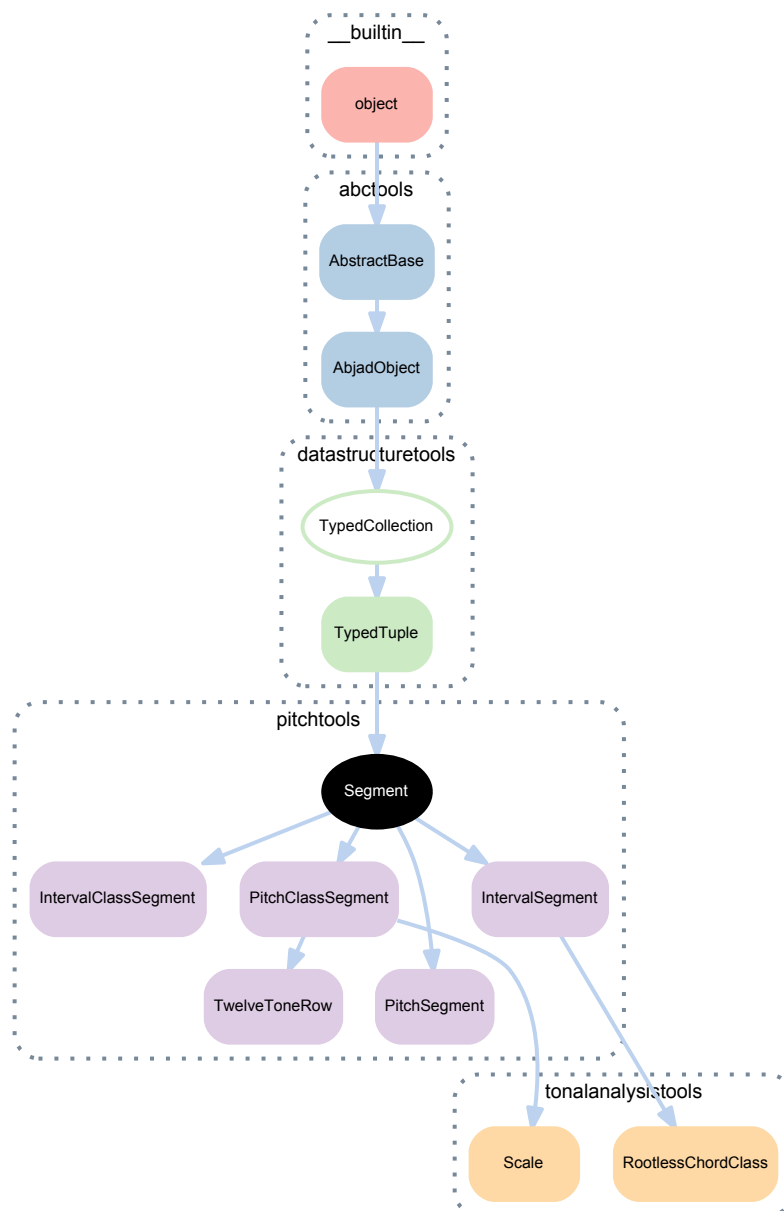
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

12.1.5 pitchtools.Segment



class `pitchtools.Segment` (*items=None, item_class=None*)
 Music-theoretic segment base class.

Bases

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

Read-only properties

`Segment`.**has_duplicates**

Is true when segment has duplicates. Otherwise false.

Returns boolean.

`(TypedCollection)`.**item_class**

Item class to coerce items into.

`(TypedCollection)`.**items**

Gets collection items.

Methods

`(TypedTuple)`.**count** (*item*)

Changes *item* to item.

Returns count in collection.

`Segment`.**from_selection** (*selection*, *item_class=None*)

Makes segment from *selection*.

Returns new segment.

`(TypedTuple)`.**index** (*item*)

Changes *item* to item.

Returns index in collection.

Special methods

`(TypedTuple)`.**__add__** (*expr*)

Adds typed tuple to *expr*.

Returns new typed tuple.

`(TypedTuple)`.**__contains__** (*item*)

Change *item* to item and return true if item exists in collection.

Returns none.

`(TypedCollection)`.**__eq__** (*expr*)

Is 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.

(TypedTuple) .**__mul__**(*expr*)
Multiplies typed tuple by *expr*.
Returns new typed tuple.

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

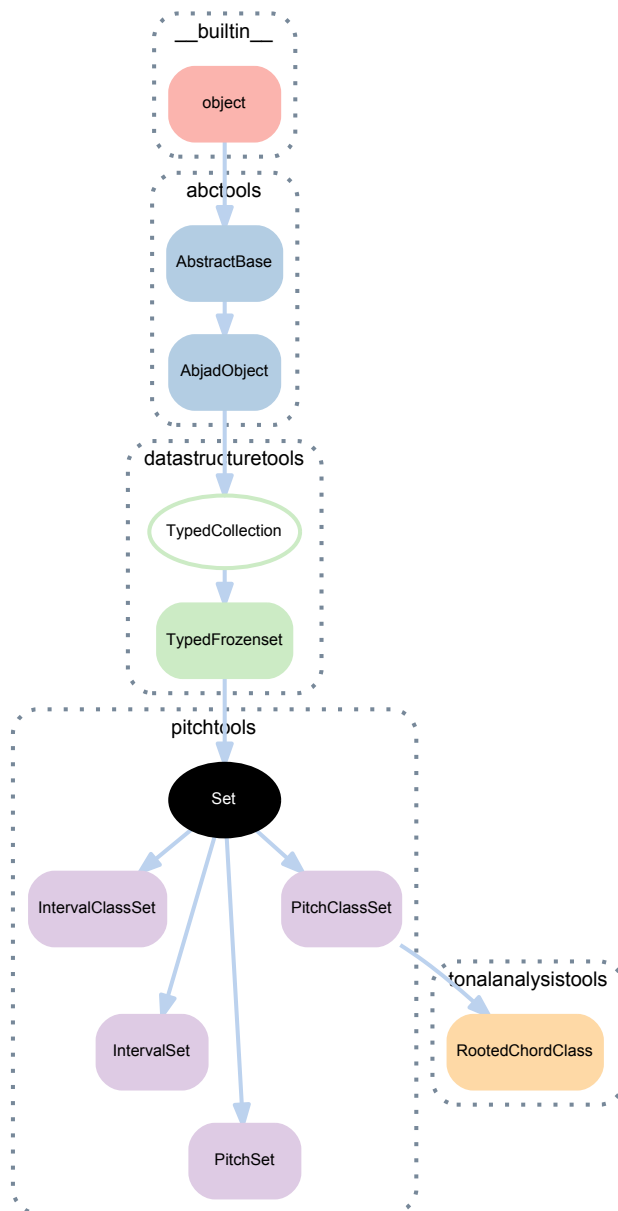
(TypedTuple) .**__radd__**(*expr*)
Right-adds *expr* to typed tuple.

(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.

12.1.6 pitchtools.Set



class `pitchtools.Set` (*items=None, item_class=None*)
 Music-theoretic set base class.

Bases

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

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)`
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)`
Is true when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

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

Returns boolean.

`(TypedFrozenSet).issuperset(expr)`
Is 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__(item)`
Is true when typed collection container *item*. Otherwise false.

Returns boolean.

(TypedCollection) .**__eq__**(*expr*)
 Is 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*)
 Is true when typed frozen set is greater than or equal to *expr*. Otherwise false.
 Returns boolean.

(TypedFrozenSet) .**__gt__**(*expr*)
 Is 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*)
 Is 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*)
 Is true when typed frozen set is less than *expr*. Otherwise false.
 Returns boolean.

(TypedFrozenSet) .**__ne__**(*expr*)
 Is 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.

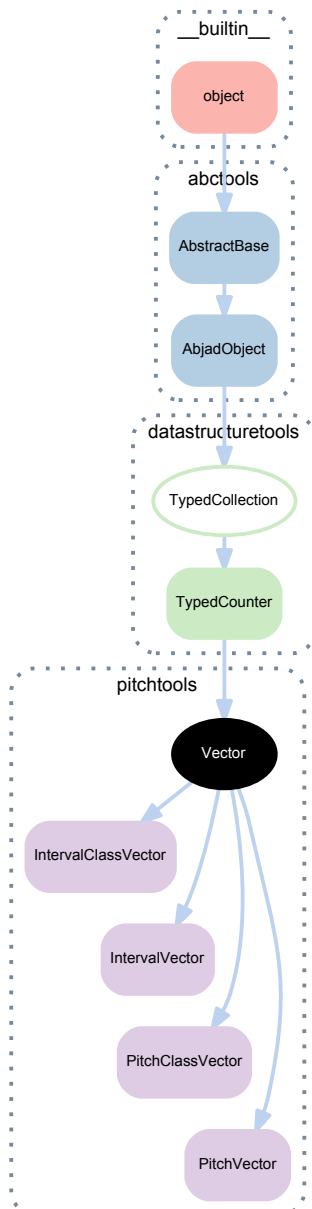
(AbjadObject) .**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

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.

12.1.7 pitchtools.Vector



class `pitchtools.Vector` (*items=None, item_class=None*)
 Music-theoretic vector base class.

Bases

- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`

- `__builtin__.object`

Read-only properties

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

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)`
Makes vector from *selection*.
Returns vector.

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

`(TypedCounter).keys()`
Iterates 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()`
Iterates values in 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__** (*item*)
Is true when typed collection container *item*. Otherwise false.
Returns boolean.

(TypedCounter) .**__delitem__** (*item*)
Deletes *item* from typed counter.
Returns none.

(TypedCollection) .**__eq__** (*expr*)
Is 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__** (*item*)
Gets *item* from typed counter.
Returns item.

(TypedCollection) .**__hash__** ()
Hashes typed collection.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
Returns integer.

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

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

(TypedCounter) .**__missing__** (*item*)
Returns zero.
Returns zero.

(TypedCollection) .**__ne__** (*expr*)
Is 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__**(*item*, *value*)

Sets typed counter *item* to *value*.

Returns none.

Vector.**__str__**()

String representation of vector.

Returns string.

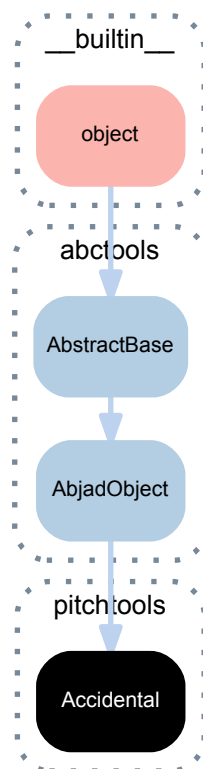
(TypedCounter).**__sub__**(*expr*)

Subtracts *expr* from typed counter.

Returns new typed counter.

12.2 Concrete classes

12.2.1 pitchtools.Accidental



class pitchtools.**Accidental** (*arg*='')

An accidental.

```
>>> pitchtools.Accidental('s')
Accidental('s')
```

Accidentals are immutable.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

Is 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*)

Is 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)`

Is true when *arg* is an accidental with an abbreviation equal to that of this accidental. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`Accidental.__ge__(arg)`

Is true when *arg* is an accidental with semitones less than or equal to those of this accidental. Otherwise false.

Returns boolean.

`Accidental.__gt__(arg)`

Is true when *arg* is an accidental with semitones less than those of this accidental. Otherwise false.

Returns boolean.

`Accidental.__hash__()`

Hashes accidental.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`Accidental.__le__(arg)`

Is true when *arg* is an accidental with semitones greater than or equal to those of this accidental. Otherwise false.

Returns boolean.

`Accidental.__lt__(arg)`

Is true when *arg* is an accidental with semitones greater than those of this accidental. Otherwise false.

Returns boolean.

`Accidental.__ne__(arg)`

Is 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.

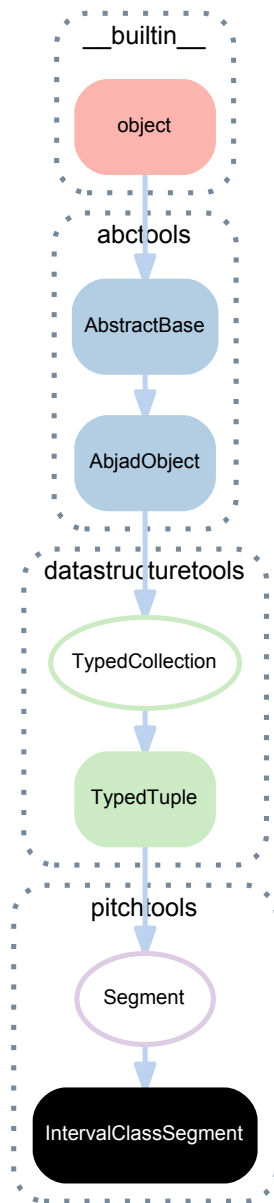
`Accidental.__str__()`

String representation of accidental.

Returns string.

`Accidental.__sub__(arg)`
 Subtracts *arg* from accidental.
 Returns new accidental.

12.2.2 `pitchtools.IntervalClassSegment`



class `pitchtools.IntervalClassSegment` (*items=None, item_class=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`
- `abctools.AbjadObject.AbstractBase`
- `__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`

Is true when all diatonic interval-classes in segment are tertian. Otherwise false:

```
>>> interval_class_segment = pitchtools.IntervalClassSegment(
...     items=[('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 items into.

`(TypedCollection).items`

Gets collection items.

Methods

`(TypedTuple).count(item)`

Changes *item* to item.

Returns count in collection.

`(TypedTuple).index(item)`

Changes *item* to item.

Returns index in collection.

Class methods

`IntervalClassSegment.from_selection(selection, item_class=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__**(*item*)
Change *item* to item and return true if item exists in collection.

Returns none.

(TypedCollection) .**__eq__**(*expr*)
Is 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.

(TypedTuple) .**__mul__**(*expr*)
Multiplies typed tuple by *expr*.

Returns new typed tuple.

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

Returns boolean.

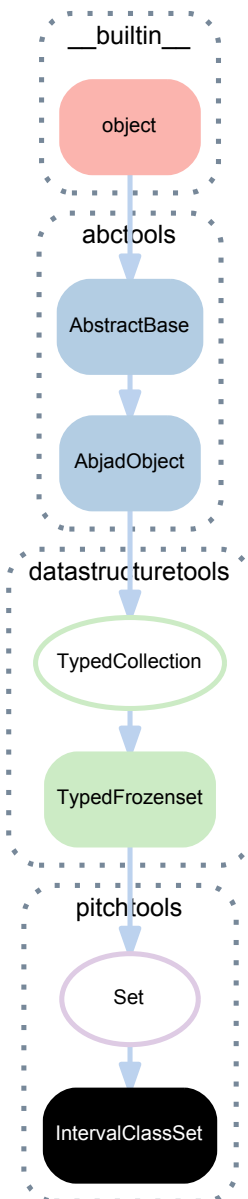
(TypedTuple) .**__radd__**(*expr*)
Right-adds *expr* to typed tuple.

`(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.

12.2.3 pitchtools.IntervalClassSet



class `pitchtools.IntervalClassSet` (*items=None, item_class=None*)
 An interval-class set.

Bases

- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

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)`
Is true when typed frozen set shares no elements with *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).issubset(expr)`
Is true when typed frozen set is a subset of *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).issuperset(expr)`
Is 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)`

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 sorted(interval_classes):
...     interval_class
...
NamedIntervalClass('-M6')
NamedIntervalClass('-P5')
NamedIntervalClass('-aug4')
NamedIntervalClass('-M3')
NamedIntervalClass('-m3')
NamedIntervalClass('-M2')
NamedIntervalClass('+m2')
NamedIntervalClass('+M2')
NamedIntervalClass('+m3')
NamedIntervalClass('+M3')
NamedIntervalClass('+P4')
NamedIntervalClass('+aug4')
NamedIntervalClass('+P5')
NamedIntervalClass('+M6')
NamedIntervalClass('+m7')
NamedIntervalClass('+M7')
NamedIntervalClass('+P8')
```

Returns interval set.

Special methods

`(TypedFrozenSet).__and__(expr)`

Logical AND of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedCollection).__contains__(item)`

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

`(TypedCollection).__eq__(expr)`

Is 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)`

Is true when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).__gt__(expr)`

Is 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*)
Is 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*)
Is true when typed frozen set is less than *expr*. Otherwise false.
Returns boolean.

(TypedFrozenSet) .**__ne__**(*expr*)
Is 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.

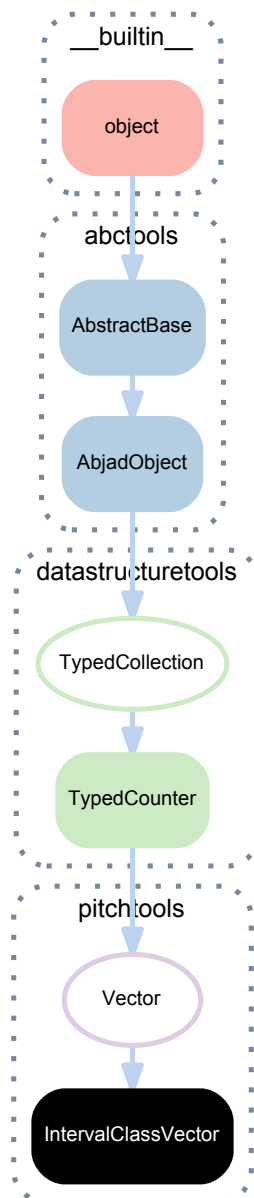
(AbjadObject) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(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.

12.2.4 pitchtools.IntervalClassVector



class `pitchtools.IntervalClassVector` (*items=None, item_class=None*)
 An interval-class vector.

```

>>> pitch_segment = pitchtools.PitchSegment(
...     items=[0, 11, 7, 4, 2, 9, 3, 8, 10, 1, 5, 6],
... )
>>> numbered_interval_class_vector = pitchtools.IntervalClassVector(
...     items=pitch_segment,
...     item_class=pitchtools.NumberedInversionEquivalentIntervalClass,
... )
>>> items = sorted(numbered_interval_class_vector.items())
>>> for interval, count in items:
...     print(interval, count)
...
1 12
2 12
3 12
4 12
5 12
6 6
  
```

Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

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()`
Iterates items in typed counter.

Yields items.

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

`(TypedCounter).viewitems()`
Please document.

`(TypedCounter).viewkeys()`
Please document.

`(TypedCounter).viewvalues()`
Please document.

Class methods

`IntervalClassVector.from_selection(selection, item_class=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__(item)`
 Is true when typed collection container *item*. Otherwise false.
 Returns boolean.

`(TypedCounter).__delitem__(item)`
 Deletes *item* from typed counter.
 Returns none.

`(TypedCollection).__eq__(expr)`
 Is 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__(item)`
 Gets *item* from typed counter.
 Returns item.

`(TypedCollection).__hash__()`
 Hashes typed collection.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 Returns integer.

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

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

`(TypedCounter).__missing__(item)`
 Returns zero.
 Returns zero.

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

Is 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__**(*item*, *value*)

Sets typed counter *item* to *value*.

Returns none.

(Vector) .**__str__**()

String representation of vector.

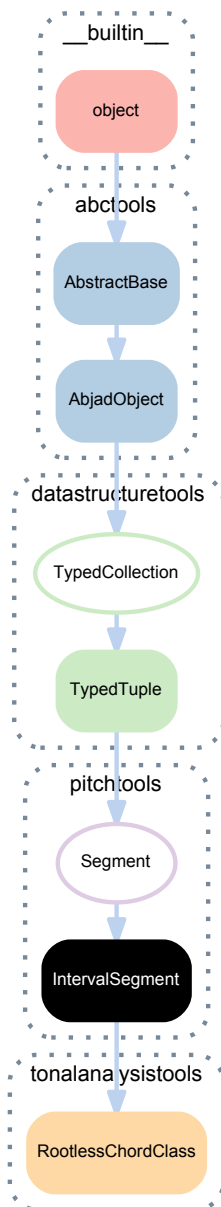
Returns string.

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

Subtracts *expr* from typed counter.

Returns new typed counter.

12.2.5 pitchtools.IntervalSegment



class `pitchtools.IntervalSegment` (*items=None, item_class=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`
- `abctools.AbjadObject.AbstractBase`
- `__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 items into.

`(TypedCollection).items`

Gets collection items.

`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.

Methods

`(TypedTuple).count(item)`

Changes *item* to item.

Returns count in collection.

`(TypedTuple).index(item)`

Changes *item* 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)`

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__**(*item*)

Change *item* to item and return true if item exists in collection.

Returns none.

(TypedCollection) .**__eq__**(*expr*)

Is 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.

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

Multiplies typed tuple by *expr*.

Returns new typed tuple.

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

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

Returns boolean.

(TypedTuple) .**__radd__**(*expr*)

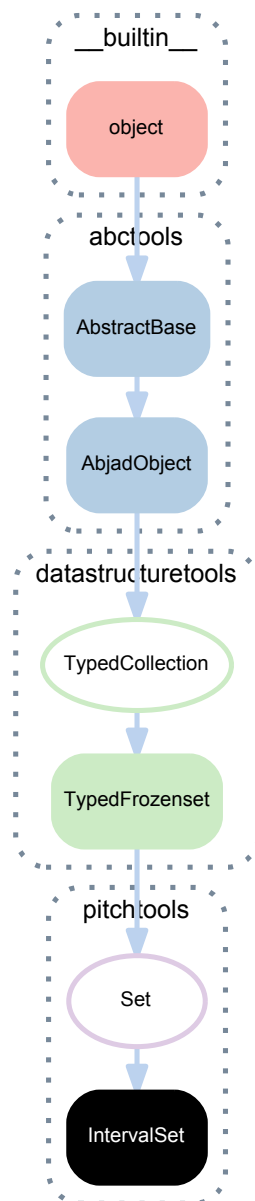
Right-adds *expr* to typed tuple.

(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.

12.2.6 pitchtools.IntervalSet



class pitchtools.**IntervalSet** (*items=None, item_class=None*)
 An interval set.

Bases

- `pitchtools.Set`
- `datastructuretools.TypedFrozenSet`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

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)`
Is true when typed frozen set shares no elements with *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).issubset(expr)`
Is true when typed frozen set is a subset of *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenSet).issuperset(expr)`
Is 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)`

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 sorted(intervals):
...     interval
...
NamedInterval('-M6')
NamedInterval('-P5')
NamedInterval('-aug4')
NamedInterval('-M3')
NamedInterval('-m3')
NamedInterval('-M2')
NamedInterval('+m2')
NamedInterval('+m3')
NamedInterval('+M3')
NamedInterval('+P4')
NamedInterval('+P5')
NamedInterval('+m7')
NamedInterval('+M7')
NamedInterval('+P8')
NamedInterval('+M9')
NamedInterval('+aug11')
NamedInterval('+M13')
```

Returns interval set.

Special methods

`(TypedFrozenSet).__and__(expr)`

Logical AND of typed frozen set and *expr*.

Returns new typed frozen set.

`(TypedCollection).__contains__(item)`

Is true when typed collection contains *item*. Otherwise false.

Returns boolean.

`(TypedCollection).__eq__(expr)`

Is 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)`

Is true when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).__gt__(expr)`

Is 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*)
Is 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*)
Is true when typed frozen set is less than *expr*. Otherwise false.
Returns boolean.

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

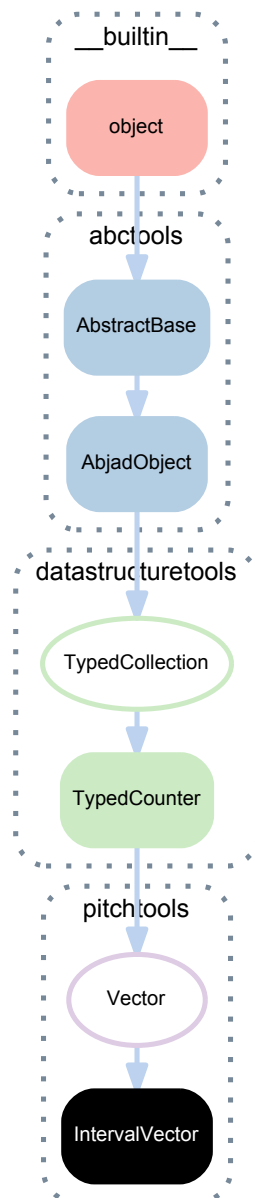
(TypedFrozenSet) .**__or__**(*expr*)
Logical OR of typed frozen set and *expr*.
Returns new typed frozen set.

(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.

12.2.7 `pitchtools.IntervalVector`

class `pitchtools.IntervalVector` (*items=None, item_class=None*)
 An interval vector.

```

>>> pitch_segment = pitchtools.PitchSegment(
...     items=[0, 11, 7, 4, 2, 9, 3, 8, 10, 1, 5, 6],
... )
>>> numbered_interval_vector = pitchtools.IntervalVector(
...     items=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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

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()`
Iterates items in typed counter.
Yields items.

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

`(TypedCounter).viewitems()`
Please document.

`(TypedCounter).viewkeys()`
Please document.

`(TypedCounter).viewvalues()`
Please document.

Class methods

`IntervalVector.from_selection(selection, item_class=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__(item)`
Is true when typed collection contains *item*. Otherwise false.

Returns boolean.

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

Returns none.

`(TypedCollection).__eq__(expr)`
Is 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__(item)`
Gets *item* from typed counter.

Returns item.

`(TypedCollection).__hash__()`
Hashes typed collection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

(TypedCounter) .**__missing__**(*item*)
Returns zero.
Returns zero.

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

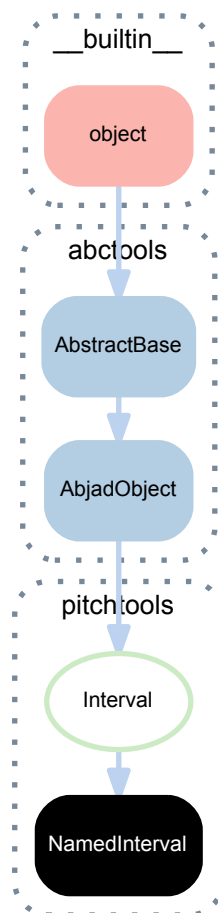
(AbjadObject) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(TypedCounter) .**__setitem__**(*item*, *value*)
Sets typed counter *item* to *value*.
Returns none.

(Vector) .**__str__**()
String representation of vector.
Returns string.

(TypedCounter) .**__sub__**(*expr*)
Subtracts *expr* from typed counter.
Returns new typed counter.

12.2.8 pitchtools.NamedInterval



class `pitchtools.NamedInterval(*args)`
 A named interval.

```
>>> interval = pitchtools.NamedInterval('+M9')
>>> interval
NamedInterval('+M9')
```

```
>>> interval = pitchtools.NamedInterval(-4)
>>> interval
NamedInterval('-M3')
```

Bases

- `pitchtools.Interval`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.octaves`
Number of octaves in interval.

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*)
Is 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*)
Is 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.

```
>>> interval = pitchtools.NamedInterval('+M9')
>>> 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)`

Is 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 Abjad 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`

`NamedInterval.__hash__()`

Hashes named interval.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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)`

Is 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)`
Is 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'
```

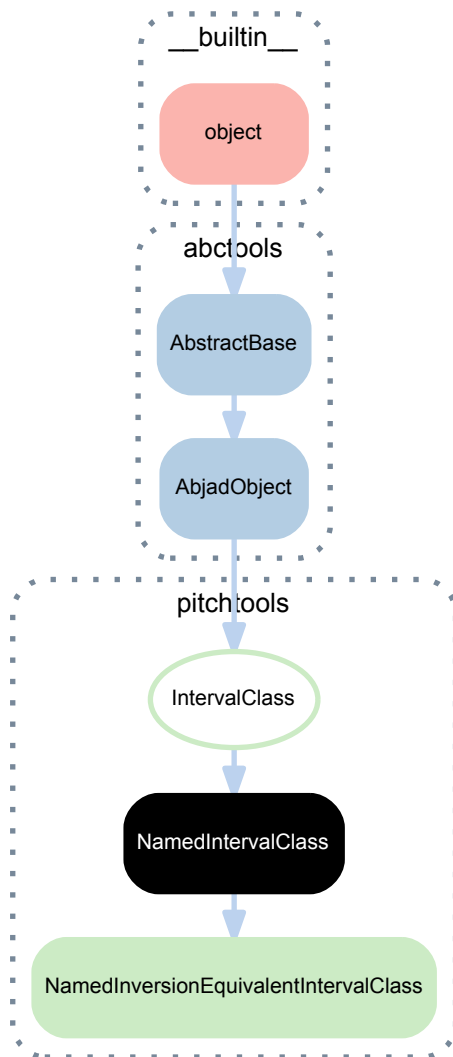
Returns string.

`NamedInterval.__sub__(arg)`
Subtracts *arg* from named interval.

```
>>> interval - pitchtools.NamedInterval('+M2')
NamedInterval('+P8')
```

Returns new named interval.

12.2.9 `pitchtools.NamedIntervalClass`



class `pitchtools.NamedIntervalClass` (*args)
 A named interval-class.

```
>>> pitchtools.NamedIntervalClass('-M9')
NamedIntervalClass('-M2')
```

Bases

- `pitchtools.IntervalClass`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`NamedIntervalClass.direction_number`
 Direction number of named interval-class.
 Returns -1, 0 or 1.

`NamedIntervalClass.direction_string`

Direction word of named interval-class.

Returns string.

`NamedIntervalClass.direction_symbol`

Direction symbol 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)`

Is 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 Abjad 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.__lt__(arg)`

Is true when *arg* is a named interval class with a number greater than that of this named interval.

`NamedIntervalClass.__ne__(arg)`

Is true when named interval-class does not equal *arg*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

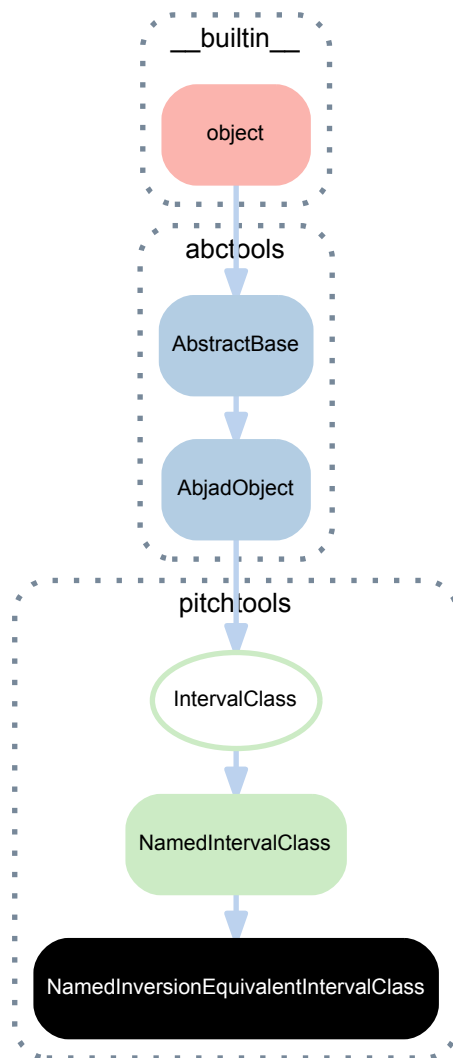
Returns string.

`NamedIntervalClass.__str__()`

String representation of named interval-class.

Returns string.

12.2.10 `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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(NamedIntervalClass).direction_number`
Direction number of named interval-class.

Returns -1, 0 or 1.

`(NamedIntervalClass).direction_string`
Direction word of named interval-class.

Returns string.

`(NamedIntervalClass).direction_symbol`
Direction symbol 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)`
Is 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 Abjad object.

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

Returns string.

NamedInversionEquivalentIntervalClass.**__hash__**()

Required to be explicitly re-defined on Python 3 if **__eq__** changes

Returns integer.

(NamedIntervalClass).**__int__**()

Changes named interval-class to integer.

Returns integer.

(NamedIntervalClass).**__lt__**(*arg*)

Is true when *arg* is a named interval class with a number greater than that of this named interval.

NamedInversionEquivalentIntervalClass.**__ne__**(*arg*)

Is true when named inversion-equivalent interval-class does not equal *arg*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

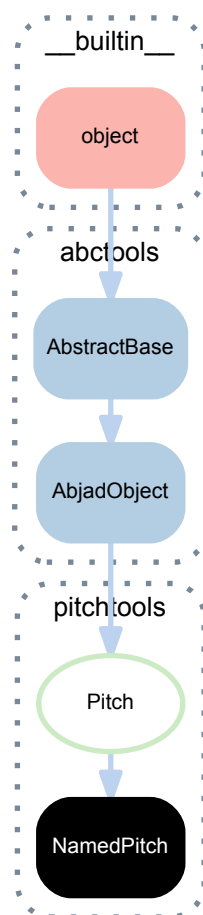
Returns string.

(NamedIntervalClass).**__str__**()

String representation of named interval-class.

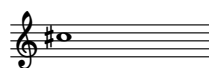
Returns string.

12.2.11 pitchtools.NamedPitch



class `pitchtools.NamedPitch(*args)`
 A named pitch.

```
>>> pitch = NamedPitch("cs' ' ")
>>> show(pitch)
```



Bases

- `pitchtools.Pitch`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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*.

```
>>> pitchtools.NamedPitch("d'").invert("c'")
NamedPitch('bf')
```

```
>>> pitchtools.NamedPitch('bf').invert("c'")
NamedPitch("d'")
```

```
>>> pitchtools.NamedPitch("d'").invert('a')
NamedPitch('e')
```

Returns new named pitch.

`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*.

```
>>> NamedPitch("c'").transpose('m2')
NamedPitch("df'")
```

```
>>> NamedPitch("c'").transpose('-M2')
NamedPitch('bf')
```

Returns new named pitch.

Class methods

`(Pitch).from_hertz (hertz)`
 Creates pitch from *hertz*.

```
>>> pitchtools.NamedPitch.from_hertz(440)
NamedPitch("a'")
```

```
>>> pitchtools.NumberedPitch.from_hertz(440)
NumberedPitch(9)
```

Returns new pitch.

Static methods

`(Pitch).is_diatonic_pitch_name(expr)`
Is 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)`
Is 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)`
Is 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)`
Is 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)`
Is 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.__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)`
Is 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)`
Is true when named pitch is greater than or equal to *arg*. Otherwise false.
Returns boolean.

`NamedPitch.__gt__(arg)`
Is true when named pitch is greater than *arg*. Otherwise false.
Returns boolean.

`NamedPitch.__hash__()`
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
Returns integer.

`(Pitch).__illustrate__()`
Illustrates pitch.
Returns LilyPond file.

`NamedPitch.__int__()`
Changes named pitch to integer.

```
>>> int(pitch)
13
```

Returns integer.

`NamedPitch.__le__(arg)`

Is true when named pitch is less than or equal to *arg*. Otherwise false.

Returns boolean.

`NamedPitch.__lt__(arg)`

Is true when named pitch is less than *arg*. Otherwise false.

Returns boolean.

`NamedPitch.__ne__(arg)`

Is 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.

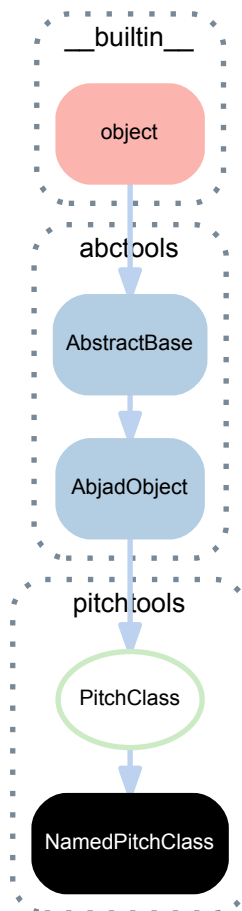
`NamedPitch.__sub__(arg)`

Subtracts *arg* from named pitch.

```
>>> NamedPitch("cs'") - NamedPitch("b'")
NamedInterval(' -M2')
```

Returns named interval.

12.2.12 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`
- `abctools.AbjadObject.AbstractBase`
- `__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(expr)`
Transposes named pitch-class by named interval *expr*.

```
>>> 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)`
Is 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)`
Is 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*)
Is 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.**__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*)
Is 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.

`NamedPitchClass.__hash__()`

Hashes named pitch-class.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

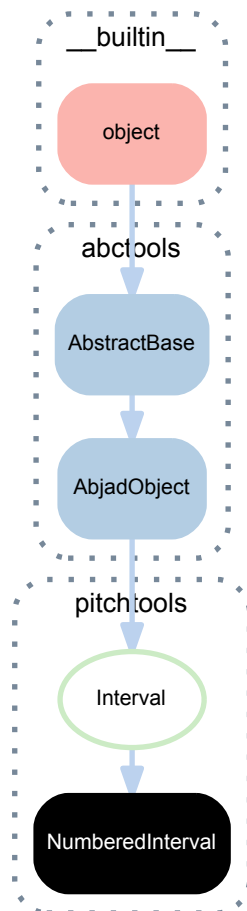
`NamedPitchClass.__sub__(arg)`

Subtracts *arg* from named pitch-class.

```
>>> pitch_class - pitchtools.NamedPitchClass('g')
NamedInversionEquivalentIntervalClass('+aug4')
```

Returns named inversion-equivalent interval-class.

12.2.13 pitchtools.NumberedInterval



class `pitchtools.NumberedInterval` (*arg=None*)
 A numbered interval.

```
>>> numbered_interval = pitchtools.NumberedInterval(-14)
>>> numbered_interval
NumberedInterval(-14)
```

Bases

- `pitchtools.Interval`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

`NumberedInterval.direction_string`
Direction string of named interval.

```
>>> pitchtools.NumberedInterval(-14).direction_string
'descending'
```

Returns 'ascending', 'descending' or none.

`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*)
Is 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*)
Is 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)`

Is 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 Abjad 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)`

Is 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)`

Is true when interval does not equal *arg*.

Returns boolean.

`NumberedInterval.__neg__()`

Negates numbered interval.

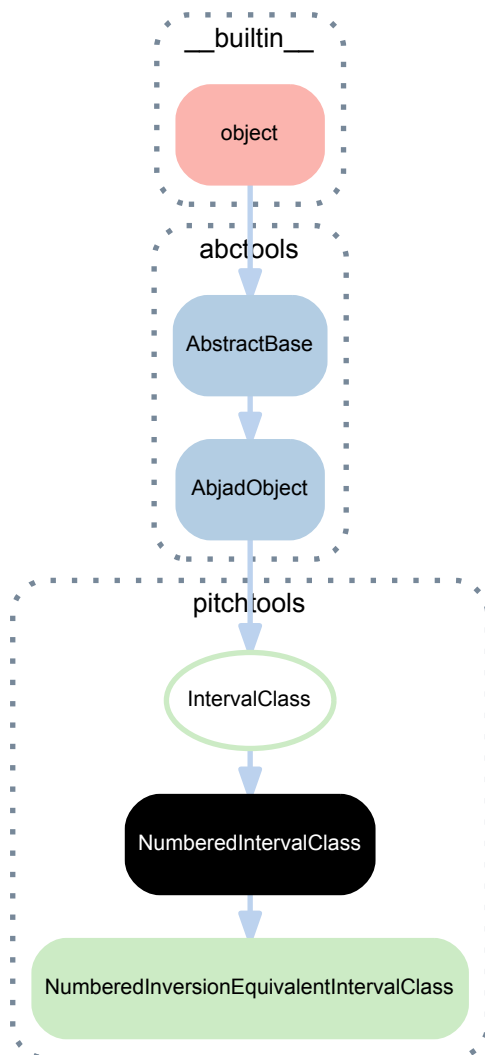
Returns new numbered interval.

`(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.

12.2.14 pitchtools.NumberedIntervalClass



class `pitchtools.NumberedIntervalClass` (*item=None*)
 A numbered interval-class.

```
>>> pitchtools.NumberedIntervalClass(-14)
NumberedIntervalClass(-2)
```

Bases

- `pitchtools.IntervalClass`

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)`
Is 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 Abjad object.

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

Returns string.

`NumberedIntervalClass.__hash__()`

Hashes numbered interval-class.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

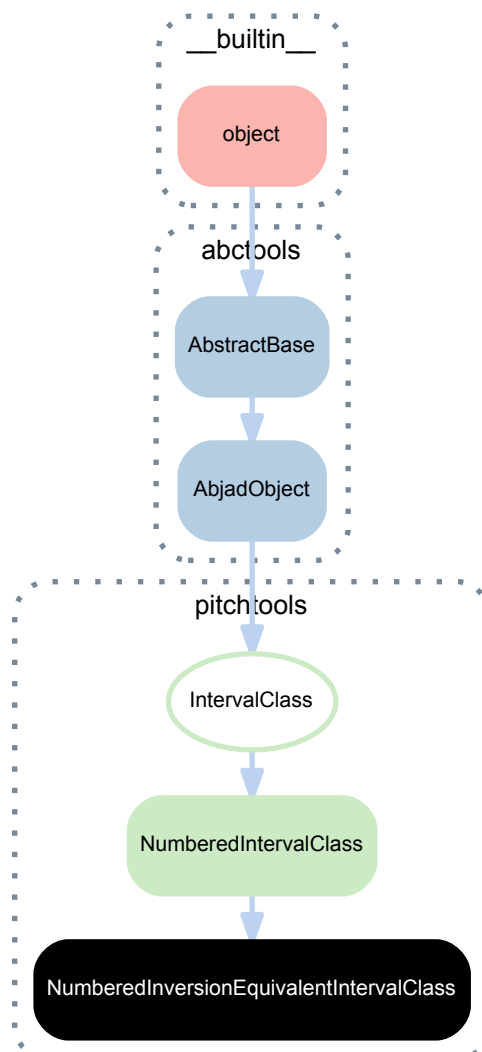
Returns string.

`(IntervalClass).__str__()`

String representation of interval-class.

Returns string.

12.2.15 `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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad 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.

`NumberedInversionEquivalentIntervalClass.__lt__(arg)`

Is true when *arg* is a numbered inversion-equivalent interval-class with a number less than this numbered inversion-equivalent interval-class.

`(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.

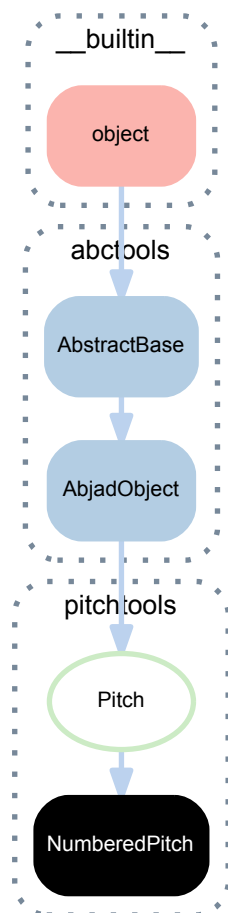
Returns string.

`NumberedInversionEquivalentIntervalClass.__str__()`

String representation of numbered inversion-equivalent interval-class.

Returns string.

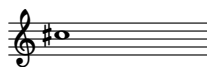
12.2.16 pitchtools.NumberedPitch



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`
- `abctools.AbjadObject.AbstractBase`
- `__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
'cs' / ''
```

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 numberd pitch around *axis*.

```
>>> pitchtools.NumberedPitch(2).invert(0)
NumberedPitch(-2)
```

```
>>> pitchtools.NumberedPitch(-2).invert(0)
NumberedPitch(2)
```

```
>>> pitchtools.NumberedPitch(2).invert(-3)
NumberedPitch(-8)
```

Returns new numbered pitch.

`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)`

Tranposes numbered pitch by *n* semitones.

```
>>> pitchtools.NumberedPitch(13).transpose(1)
NumberedPitch(14)
```

Returns new numbered pitch.

Class methods

`(Pitch).from_hertz (hertz)`

Creates pitch from *hertz*.

```
>>> pitchtools.NamedPitch.from_hertz(440)
NamedPitch("a")
```

```
>>> pitchtools.NumberedPitch.from_hertz(440)
NumberedPitch(9)
```

Returns new pitch.

Static methods

`(Pitch).is_diatonic_pitch_name (expr)`

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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 **.__add__** (*arg*)

Adds *arg* to numberd pitch.

Returns new numbered pitch.

NumberedPitch **.__eq__** (*arg*)

Is true when *arg* can be coerced to a numbered pitch and when this numbered pitch equals *arg*. 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)`
 Is true when *arg* can be coerced to a numbered pitch and when this numbered pitch is less than *arg*. Otherwise false.
 Returns boolean.

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

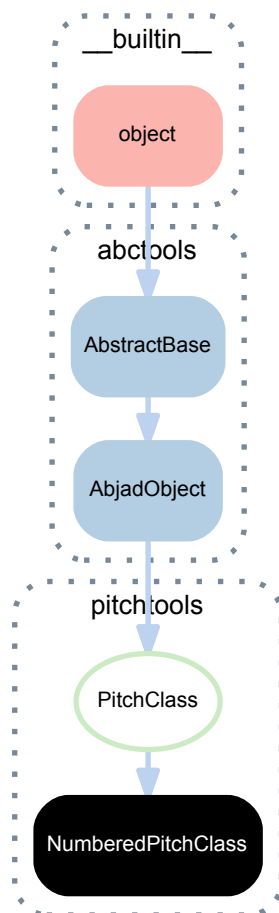
`NumberedPitch.__neg__()`
 Negates numbered pitch.
 Returns new numbered pitch.

`(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.

12.2.17 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`

- `abctools.AbjadObject.AbstractBase`
- `__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)`
Is 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)`
Is 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)`
 Is 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.__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)`
 Is 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.

`NumberedPitchClass.__hash__()`

Hashes numbered pitch-class.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

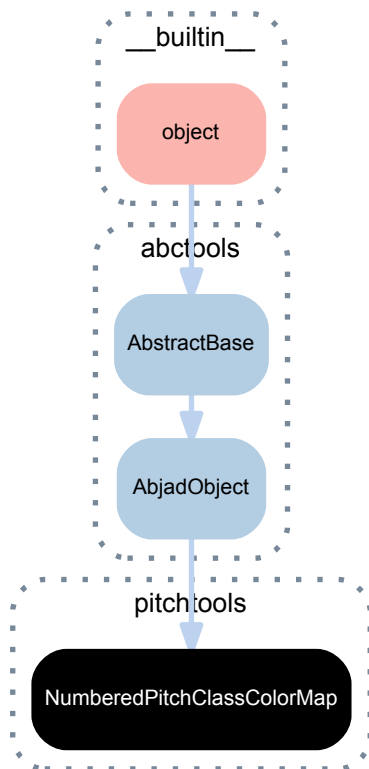
`NumberedPitchClass.__sub__(expr)`

Subtracts *expr* from numbered pitch-class.

Subtraction defined against both numbered intervals and against other pitch-classes.

Returns numbered inversion-equivalent interval-class.

12.2.18 `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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`NumberedPitchClassColorMap.colors`
 Colors of color map.

```

>>> color_map.colors
['red', 'green', 'blue']
  
```

Returns list.

`NumberedPitchClassColorMap.is_twelve_tone_complete`
 Is 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**
Is 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 Abjad 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).__hash__()`
 Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

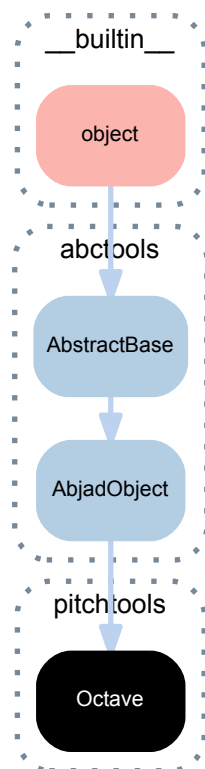
`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

Returns string.

12.2.19 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`
- `abctools.AbjadObject.AbstractBase`
- `__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(range_string='[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)`

Is 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 Abjad 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.

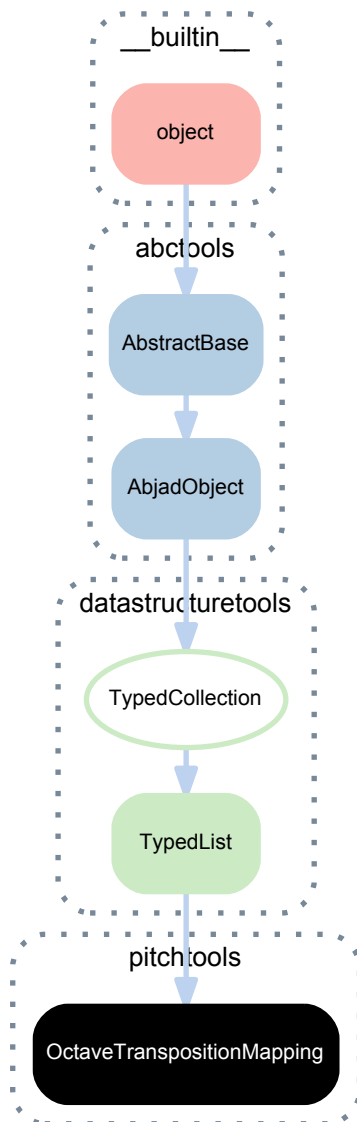
```
>>> str(pitchtools.Octave(4))
"4"
```

```
>>> str(pitchtools.Octave(1))
'1'
```

```
>>> str(pitchtools.Octave(3))
''
```

Returns string.

12.2.20 pitchtools.OctaveTranspositionMapping



class `pitchtools.OctaveTranspositionMapping` (*items=None*, *item_class=None*,
keep_sorted=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`

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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(item)`
Changes *item* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *item*)
Changes *item* 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** (*item*)
Changes *item* 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__** (*item*)
Is true when typed collection container *item*. Otherwise false.
Returns boolean.

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

Returns none.

(TypedCollection).**__eq__**(*expr*)
Is 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(
      source_pitch_range=pitchtools.PitchRange(
        range_string='[A0, C4]',
      ),
      target_octave_start_pitch=pitchtools.NumberedPitch(15),
    ),
    pitchtools.OctaveTranspositionMappingComponent(
      source_pitch_range=pitchtools.PitchRange(
        range_string='[C4, C8]',
      ),
      target_octave_start_pitch=pitchtools.NumberedPitch(27),
    ),
  ]
)
```

Returns string.

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

Returns item.

(TypedCollection).**__hash__**()
Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedList).**__iadd__**(*expr*)
Changes items 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(
      name='ppp',
    ),
    indicatortools.Dynamic(
      name='p',
    ),
    indicatortools.Dynamic(
      name='mp',
    ),
    indicatortools.Dynamic(
      name='mf',
    ),
    indicatortools.Dynamic(

```

```

        name='fff',
    ),
],
item_class=indicatortools.Dynamic,
)

```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__**().

Returns generator.

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

Changes items 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,
)

```

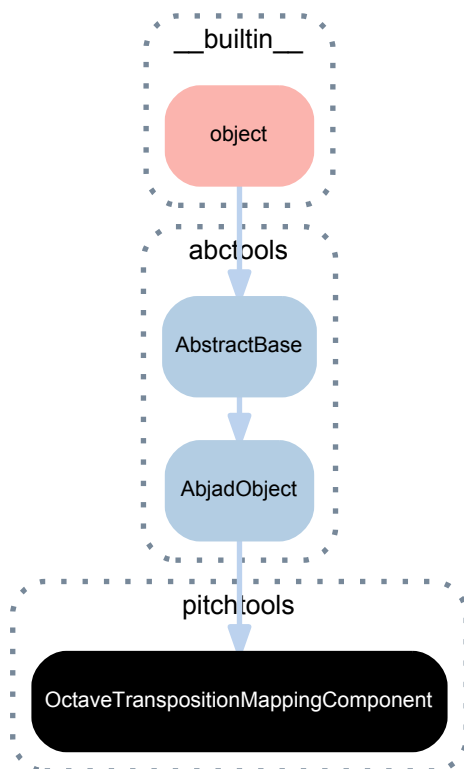
```

>>> 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,
)

```

Returns none.

12.2.21 pitchtools.OctaveTranspositionMappingComponent



class `pitchtools.OctaveTranspositionMappingComponent` (*source_pitch_range*='[A0, C8]', *target_octave_start_pitch*=0)

An octave transposition mapping component.

```
>>> mc = pitchtools.OctaveTranspositionMappingComponent(' [A0, C8]', 15)
>>> mc
OctaveTranspositionMappingComponent(source_pitch_range=PitchRange(range_string=' [A0, C8]'), target_octave_start_pitch=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.

Todo

make components immutable.

Bases

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

Read/write properties

`OctaveTranspositionMappingComponent.source_pitch_range`
Gets and sets source pitch range of mapping component.


```
>>> mc.source_pitch_range
PitchRange(range_string='[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)`

Is 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.__hash__()`

Hashes octave transposition mapping component.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

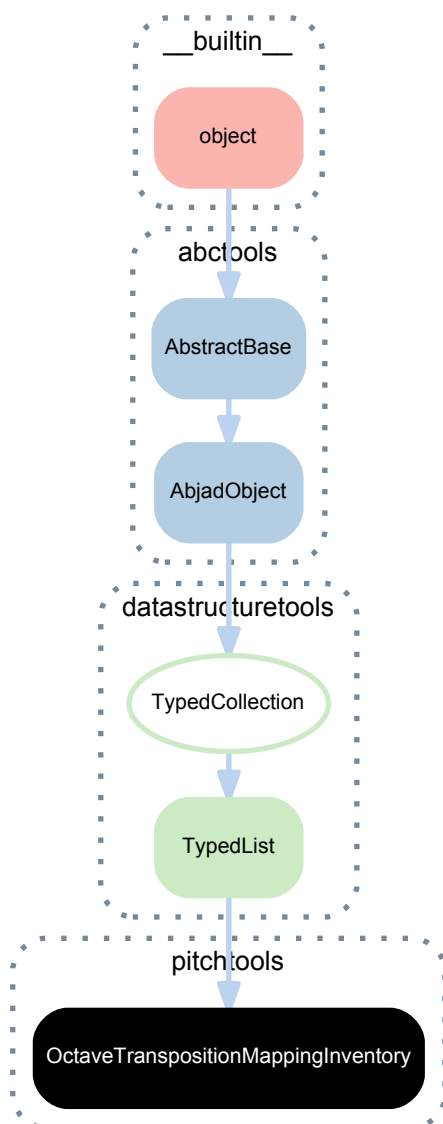
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

12.2.22 pitchtools.OctaveTranspositionMappingInventory



class `pitchtools.OctaveTranspositionMappingInventory` (*items=None*,
item_class=None,
keep_sorted=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(
                source_pitch_range=pitchtools.PitchRange(
                    range_string='A0, C4',
                ),
                target_octave_start_pitch=pitchtools.NumberedPitch(15),
            ),
            pitchtools.OctaveTranspositionMappingComponent(
```

```

        source_pitch_range=pitchtools.PitchRange(
            range_string='[C4, C8]',
        ),
        target_octave_start_pitch=pitchtools.NumberedPitch(27),
    ),
]
),
pitchtools.OctaveTranspositionMapping(
    [
        pitchtools.OctaveTranspositionMappingComponent(
            source_pitch_range=pitchtools.PitchRange(
                range_string='[A0, C8]',
            ),
            target_octave_start_pitch=pitchtools.NumberedPitch(-18),
        ),
    ]
),
]
)

```

Octave transposition mapping inventories implement list interface and are mutable.

Bases

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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** (*item*)

Changes *item* to item and returns count.

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

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

Returns count.

(TypedList) **.extend** (*items*)

Changes *items* 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** (*item*)

Changes *item* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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__** (item)
Is true when typed collection container *item*. Otherwise false.
Returns boolean.

(TypedList) .**__delitem__** (i)
Aliases list.__delitem__().
Returns none.

(TypedCollection) .**__eq__** (expr)
Is 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__().
Returns item.

(TypedCollection) .**__hash__** ()
Hashes typed collection.
Required to be explicitly re-defined on Python 3 if __eq__ changes.
Returns integer.

(TypedList) .**__iadd__** (expr)
Changes items 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(
        name='ppp',
    ),
    indicatortools.Dynamic(
        name='p',
    ),
    indicatortools.Dynamic(
        name='mp',
    ),
    indicatortools.Dynamic(
        name='mf',
    ),
    indicatortools.Dynamic(
        name='fff',
```

```

        ),
    ],
    item_class=indicatortools.Dynamic,
)

```

Returns collection.

(TypedCollection) .**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__().

Returns generator.

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

Changes items 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,
)

```

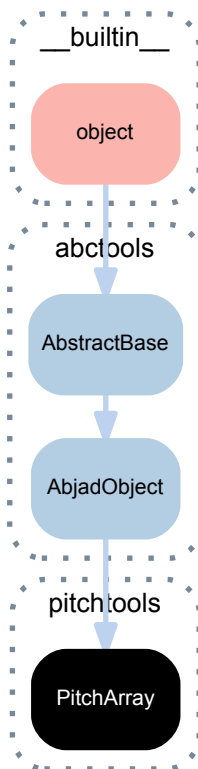
```

>>> 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,
)

```

Returns none.

12.2.23 pitchtools.PitchArray



class `pitchtools.PitchArray(*args)`
 A two-dimensional array of pitches.

Bases

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

Read-only properties

`PitchArray.cell_tokens_by_row`
 Cells items 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

Is true when pitch array has voice crossing. Otherwise false.

Returns boolean.

PitchArray.is_rectangular

Is 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)`

Is 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 = pitchtools.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 = pitchtools.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 = pitchtools.PitchArray.from_score(
...     score, populate=False)
```

```
>>> print(array)
[ ] [ ] [ ] [ ]
```


`PitchArray.__hash__()`

Hashes pitch array.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`PitchArray.__iadd__(arg)`

Adds *arg* to pitch array in place.

```
>>> array_1 = pitchtools.PitchArray([[1, 2, 1], [2, 1, 1]])
>>> print(array_1)
[ ] [ ] [ ]
[ ] [ ] [ ]
```

```
>>> array_2 = pitchtools.PitchArray([[3, 4], [4, 3]])
>>> print(array_2)
[ ] [ ]
[ ] [ ]
```

```
>>> array_3 = pitchtools.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)`

Is true when pitch array does not equal *arg*. Otherwise false.

Returns boolean.

`PitchArray.__repr__()`

Interpreter representation of pitch array.

Returns string.

`PitchArray.__setitem__(i, arg)`

Sets pitch array row *i* to *arg*.

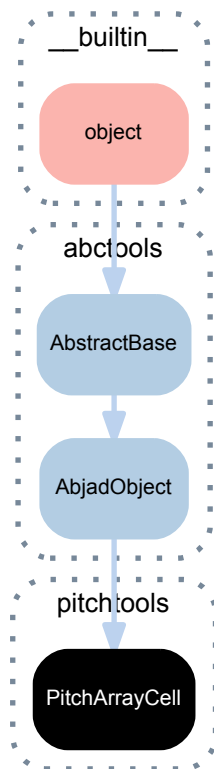
Returns none.

`PitchArray.__str__()`

String representation of pitch array.

Returns string.

12.2.24 pitchtools.PitchArrayCell



class pitchtools.**PitchArrayCell** (*item=None*)

One cell in a pitch array.

```
>>> array = pitchtools.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.item  
2
```

```
>>> cell.width  
2
```

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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`

Is true when pitch array cell is first in row. Otherwise false.

Returns boolean.

`PitchArrayCell.is_last_in_row`

Is true when pitch array cell is last in row. Otherwise false.

Returns boolean.

`PitchArrayCell.item`

Token of pitch array cell.

`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 array cell.

`PitchArrayCell.row_index`

Row index of pitch array cell.

Returns nonnegative integer or none.

`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*)

Is 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 Abjad object.

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

Returns string.

(`AbjadObject`).`__hash__` ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

Returns boolean.

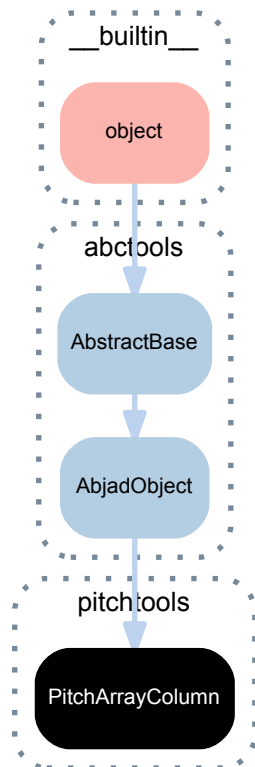
`PitchArrayCell.__repr__` ()

Gets interpreter representation of pitch array cell.

Returns string.

`PitchArrayCell.__str__()`
 String representation of pitch array cell.
 Returns string.

12.2.25 `pitchtools.PitchArrayColumn`



class `pitchtools.PitchArrayColumn` (*cells=None*)
 Column in a pitch array.

```
>>> array = pitchtools.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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`PitchArrayColumn.cell_tokens`

Cells items 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`

Is true when pitch array column has voice crossing. Otherwise false.

Returns boolean.

`PitchArrayColumn.is_defective`

Is 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)`

Is 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 Abjad 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 array cell.

`PitchArrayColumn.__hash__()`

Hashes pitch array column.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`PitchArrayColumn.__ne__(arg)`

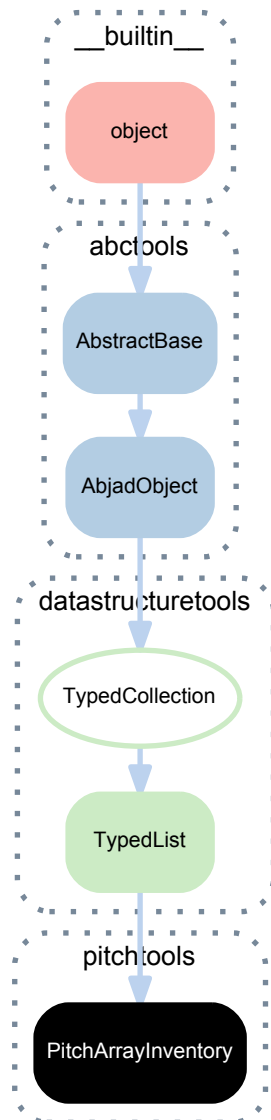
Is true when pitch array column does not equal *arg*. Otherwise false.

Returns boolean.

`PitchArrayColumn.__repr__()`
 Gets interpreter representation of pitch array column.
 Returns string.

`PitchArrayColumn.__str__()`
 String representation of pitch array column.
 Returns string.

12.2.26 pitchtools.PitchArrayInventory



class `pitchtools.PitchArrayInventory` (*items=None, item_class=None, keep_sorted=None*)
 Ordered collection of pitch arrays.

```
>>> array_1 = pitchtools.PitchArray([
...     [1, (2, 1), ([-2, -1.5], 2)],
...     [(7, 2), (6, 1), 1]])
```

```
>>> array_2 = pitchtools.PitchArray([
...     [1, 1, 1],
...     [1, 1, 1]])
```

```
>>> arrays = [array_1, array_2]
>>> inventory = pitchtools.PitchArrayInventory(arrays)
```

```
>>> print(format(inventory))
pitchtools.PitchArrayInventory(
  [
    pitchtools.PitchArray(),
    pitchtools.PitchArray(),
  ]
)
```

Bases

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

(TypedList) **.extend** (*items*)

Changes *items* 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** (*item*)

Changes *item* to item and returns index.

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

Returns index.

(TypedList) **.insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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()`

Makes score from pitch arrays in inventory.

```
>>> array_1 = pitchtools.PitchArray([
...     [1, (2, 1), (-2, -1.5), 2)],
...     [(7, 2), (6, 1), 1]])
```

```
>>> array_2 = pitchtools.PitchArray([
...     [1, 1, 1],
...     [1, 1, 1]])
```

```
>>> arrays = [array_1, array_2]
>>> inventory = pitchtools.PitchArrayInventory(arrays)
```

```
>>> score = inventory.to_score()
```

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



Create one staff per pitch-array row.

Returns score.

Special methods

`(TypedCollection).__contains__(item)`

Is true when typed collection contains *item*. Otherwise false.

Returns boolean.

`(TypedList).__delitem__(i)`

Aliases `list.__delitem__()`.

Returns none.

`(TypedCollection).__eq__(expr)`

Is 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__()`.

Returns item.

`(TypedCollection).__hash__()`

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(TypedList).__iadd__(expr)`

Changes items 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(
        name='ppp',
    ),
    indicatortools.Dynamic(
        name='p',
    ),
    indicatortools.Dynamic(
        name='mp',
    ),
    indicatortools.Dynamic(
        name='mf',
    ),
    indicatortools.Dynamic(
        name='fff',
    ),
],
item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__**().

Returns generator.

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

Changes items in *expr* to items and sets.

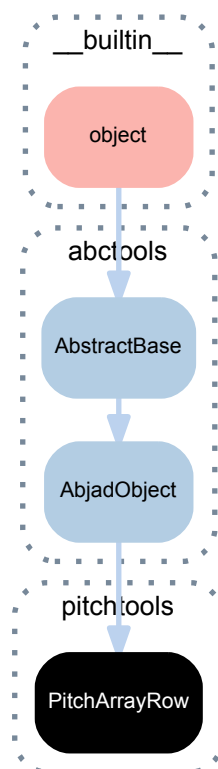
```
>>> pitch_collection[-1] = 'gqs,'
>>> print(format(pitch_collection))
datastructuretools.TypedList(
[
    pitchtools.NamedPitch("c"),
    pitchtools.NamedPitch("d"),
    pitchtools.NamedPitch("e"),
    pitchtools.NamedPitch('gqs,'),
],
item_class=pitchtools.NamedPitch,
)
```

```
>>> 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,
)
```

Returns none.

12.2.27 pitchtools.PitchArrayRow



class pitchtools.**PitchArrayRow** (*cells=None*)
A pitch array row.

```
>>> array = pitchtools.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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`PitchArrayRow.cell_tokens`

Cell items 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`

Is true when width of pitch array row does not equal width of parent pitch array. Otherwise false.

Returns boolean.

`PitchArrayRow.is_in_range`

Is 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*)

Is 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 = pitchtools.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)`

Is 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 Abjad 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.__hash__()`

Hashes pitch array row.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`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)`

Is true when pitch array row does not equal *arg*. Otherwise false.

Returns boolean.

`PitchArrayRow.__repr__()`

Gets interpreter representation of pitch array row.

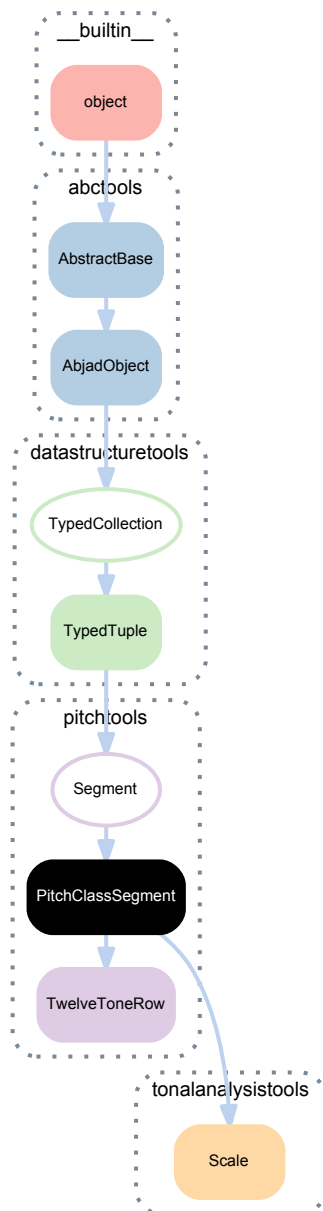
Returns string.

`PitchArrayRow.__str__()`

String representation of pitch array row.

Returns string.

12.2.28 pitchtools.PitchClassSegment



class `pitchtools.PitchClassSegment` (*items*=(-2, -1.5, 6, 7, -1.5, 7), *item_class*=None)
 A pitch-class segment.

```
>>> numbered_pitch_class_segment = pitchtools.PitchClassSegment(
...     items=[-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(
...     items=['c', 'ef', 'bqs', 'd'],
...     item_class=pitchtools.NamedPitchClass,
... )
>>> named_pitch_class_segment
PitchClassSegment(['c', 'ef', 'bqs', 'd'])
```

Pitch-class segments are immutable.

Bases

- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`PitchClassSegment`.**`has_duplicates`**

True if segment contains duplicate items:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.has_duplicates
True
```

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     items="c d e f g a b",
...     )
>>> pitch_class_segment.has_duplicates
False
```

Returns boolean.

(`TypedCollection`).**`item_class`**

Item class to coerce items into.

(`TypedCollection`).**`items`**

Gets collection items.

Methods

`PitchClassSegment`.**`alpha()`**

Morris alpha transform of pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     items=[-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`** (*item*)

Changes *item* to item.

Returns count in collection.

(`TypedTuple`).**`index`** (*item*)

Changes *item* to item.

Returns index in collection.

`PitchClassSegment`.**`invert()`**

Invert pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(
...     items=[-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)`

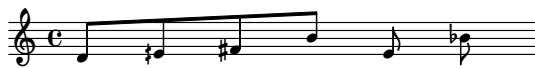
Make first *n* notes in pitch class segment.

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(
...     items=[-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(
...     items=[-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, transpose=False)`

Rotate pitch-class segment:

```
>>> pitchtools.PitchClassSegment(
...     items=[-2, -1.5, 6, 7, -1.5, 7],
```

```
...     ).rotate(1)
PitchClassSegment([7, 10, 10.5, 6, 7, 10.5])
```

```
>>> pitchtools.PitchClassSegment(
...     items=['c', 'ef', 'bqs', 'd'],
...     ).rotate(-2)
PitchClassSegment(['bqs', 'd', 'c', 'ef'])
```

If *transpose* is true, transpose the rotated segment to begin at the same pitch class as this segment:

```
>>> pitchtools.PitchClassSegment(
...     items=['c', 'b', 'd']
...     ).rotate(1, transpose=True)
PitchClassSegment(['c', 'bf', 'a'])
```

Returns new pitch-class segment.

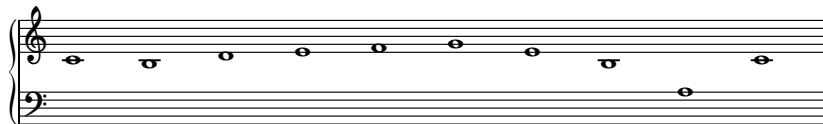
`PitchClassSegment.transpose` (*expr*)
Transpose pitch-class segment:

```
>>> pitchtools.PitchClassSegment(
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     ).transpose(10)
PitchClassSegment([8, 8.5, 4, 5, 8.5, 5])
```

Returns new pitch-class segment.

`PitchClassSegment.voice_horizontally` (*initial_octave=4*)
Voices pitch-class segment as pitch segment, with each pitch as close in distance to the previous pitch as possible.

```
>>> pitch_classes = pitchtools.PitchClassSegment(
...     "c b d e f g e b a c")
>>> pitch_segment = pitch_classes.voice_horizontally()
>>> show(pitch_segment)
```



Returns pitch segment.

`PitchClassSegment.voice_vertically` (*initial_octave=4*)
Voices pitch-class segment as pitch segment, with each pitch always higher than the previous.

```
>>> scale_degree_numbers = [1, 3, 5, 7, 9, 11, 13]
>>> scale = tonalanalysistools.Scale('c', 'minor')
>>> pitch_classes = pitchtools.PitchClassSegment((
...     scale.scale_degree_to_named_pitch_class(x)
...     for x in scale_degree_numbers))
>>> pitch_segment = pitch_classes.voice_vertically()
>>> pitch_segment
PitchSegment(['c', 'ef', 'g', 'bf', 'd', 'f', 'af'])
>>> show(pitch_segment)
```



Returns pitch segment.

Class methods

`PitchClassSegment.from_selection` (*selection, item_class=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__**(*item*)

Change *item* to item and return true if item exists in collection.

Returns none.

(TypedCollection) .**__eq__**(*expr*)

Is 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.

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

Multiplies typed tuple by *expr*.

Returns new typed tuple.

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

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

Returns boolean.

(TypedTuple) .**__radd__**(*expr*)

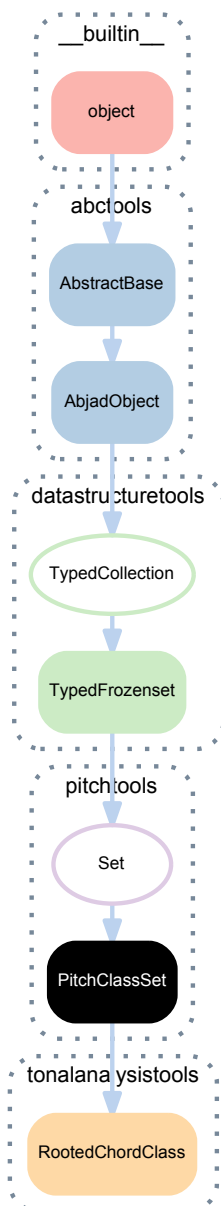
Right-adds *expr* to typed tuple.

(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.

12.2.29 pitchtools.PitchClassSet



class pitchtools.**PitchClassSet** (*items=None, item_class=None*)
 A pitch-class set.

```
>>> numbered_pitch_class_set = pitchtools.PitchClassSet(
...     items=[-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(
...     items=['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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

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)`
Is 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)`

Is 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)`

Is true when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenset).issubset(expr)`

Is true when typed frozen set is a subset of *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenset).issuperset(expr)`

Is 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)`
 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__(item)`
 Is true when typed collection contains *item*. Otherwise false.

Returns boolean.

`(TypedCollection).__eq__(expr)`
 Is 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)`
 Is true when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).__gt__(expr)`
 Is 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)`
 Is 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)`
Is true when typed frozen set is less than *expr*. Otherwise false.
Returns boolean.

`(TypedFrozenset) .__ne__(expr)`
Is 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.

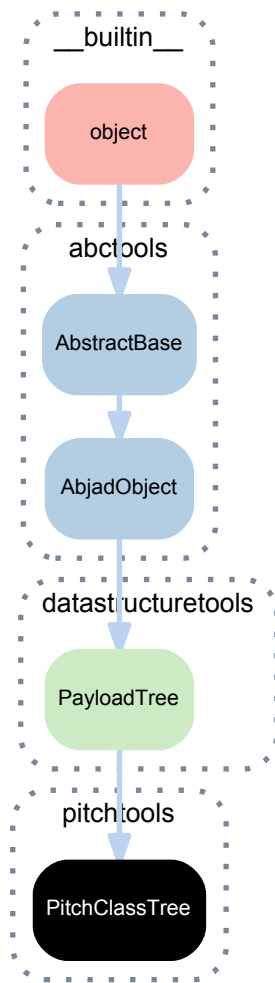
`(AbjadObject) .__repr__()`
Gets interpreter representation of Abjad object.
Returns string.

`(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.

12.2.30 pitchtools.PitchClassTree



class `pitchtools.PitchClassTree` (*items=None, item_class=None*)

A pitch-class tree.

Numbered pitch-class tree:

```

>>> tree = pitchtools.PitchClassTree(
...     items=[[0, 4, 7, 8], [9, 2, 3, 11]],
...     item_class=pitchtools.NumberedPitchClass,
... )
>>> print(format(tree, 'storage'))
pitchtools.PitchClassTree(
[
    [
        pitchtools.NumberedPitchClass(0),
        pitchtools.NumberedPitchClass(4),
        pitchtools.NumberedPitchClass(7),
        pitchtools.NumberedPitchClass(8),
    ],
    [
        pitchtools.NumberedPitchClass(9),
        pitchtools.NumberedPitchClass(2),
        pitchtools.NumberedPitchClass(3),
        pitchtools.NumberedPitchClass(11),
    ],
]
)
  
```

Named pitch-class tree:

```
>>> tree = pitchtools.PitchClassTree(
...     items=[['c', 'e', 'g', 'af'], ['a', 'd', 'ef', 'b']],
...     item_class=pitchtools.NamedPitchClass,
... )
>>> print(format(tree, 'storage'))
pitchtools.PitchClassTree(
  [
    [
      pitchtools.NamedPitchClass('c'),
      pitchtools.NamedPitchClass('e'),
      pitchtools.NamedPitchClass('g'),
      pitchtools.NamedPitchClass('af'),
    ],
    [
      pitchtools.NamedPitchClass('a'),
      pitchtools.NamedPitchClass('d'),
      pitchtools.NamedPitchClass('ef'),
      pitchtools.NamedPitchClass('b'),
    ],
  ]
)
```

Pitch-class trees are treated as immutable.

Bases

- `datastructuretools.PayloadTree`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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).expr`

Gets input argument.

`(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 Graphviz representation of payload tree.

```

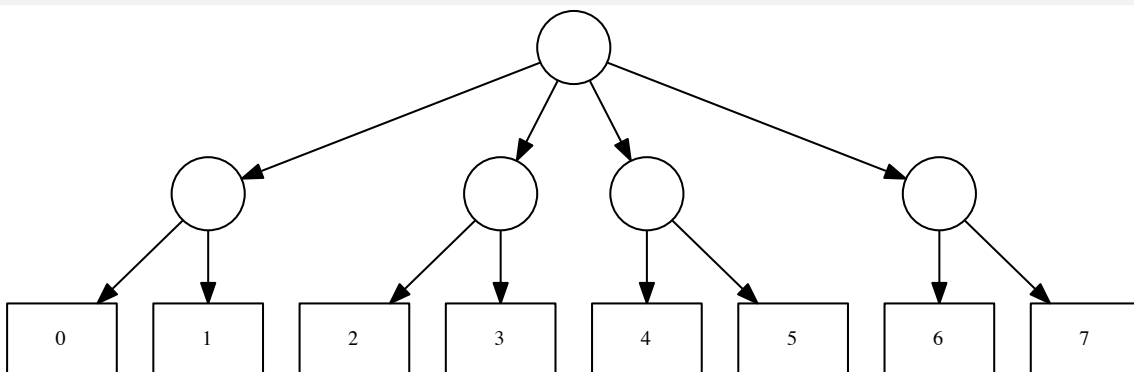
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = datastructuretools.PayloadTree(sequence)

```

```

>>> 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) **.item_class**
Gets item class of payload tree.

```
>>> tree.item_class is None
True
```

Set item class to coerce input at initialization:

```
>>> tree = datastructuretools.PayloadTree(
...     expr=[[1.1, 2.2], [8.8, 9.9]],
...     item_class=int,
... )
>>> tree
PayloadTree([[1, 2], [8, 9]])
```

Returns class or none.

(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.

Payload tree of length greater than 1 for examples with positive *n*:

```
>>> 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])]
```

Payload tree of length greater than 1 for examples with negative n :

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

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*)

Is 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)`

Is 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)`

Is 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).__hash__()`

Hashes payload tree.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`PitchClassTree.__illustrate__(**kwargs)`
Illustrates pitch-class tree.

Returns LilyPond file.

`(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.

Typical payload tree:

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

Payload tree leaf:

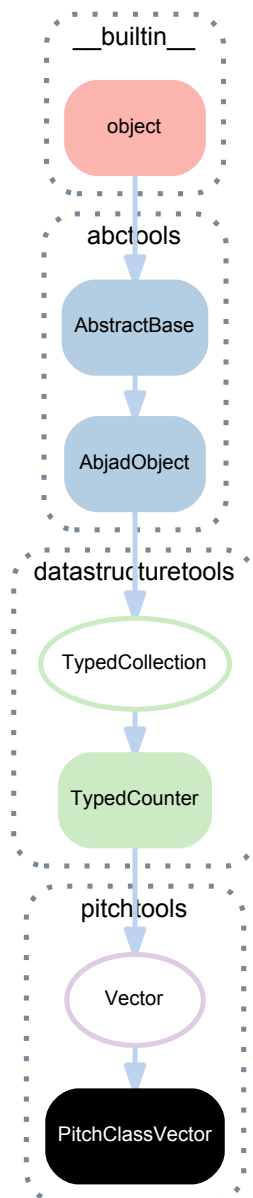
```
>>> datastructuretools.PayloadTree(0)
PayloadTree(0)
```

Empty payload tree:

```
>>> datastructuretools.PayloadTree()
PayloadTree([])
```

Returns string.

12.2.31 `pitchtools.PitchClassVector`



class `pitchtools.PitchClassVector` (*items=None, item_class=None*)
 A pitch-class vector.

Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

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()`
Iterates items in typed counter.
Yields items.

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

`(TypedCounter).viewitems()`
Please document.

`(TypedCounter).viewkeys()`
Please document.

`(TypedCounter).viewvalues()`
Please document.

Class methods

`PitchClassVector.from_selection(selection, item_class=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__** (*item*)
 Is true when typed collection container *item*. Otherwise false.
 Returns boolean.

(TypedCounter) .**__delitem__** (*item*)
 Deletes *item* from typed counter.
 Returns none.

(TypedCollection) .**__eq__** (*expr*)
 Is 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__** (*item*)
 Gets *item* from typed counter.
 Returns item.

(TypedCollection) .**__hash__** ()
 Hashes typed collection.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 Returns integer.

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

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

(TypedCounter) .**__missing__** (*item*)
 Returns zero.
 Returns zero.

(TypedCollection) .**__ne__** (*expr*)
 Is 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__**(*item*, *value*)

Sets typed counter *item* to *value*.

Returns none.

(Vector).**__str__**()

String representation of vector.

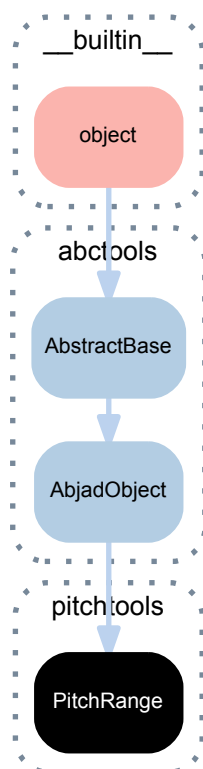
Returns string.

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

Subtracts *expr* from typed counter.

Returns new typed counter.

12.2.32 pitchtools.PitchRange

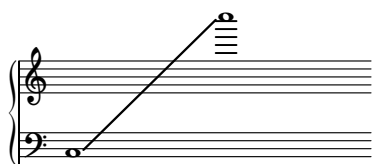


```
class pitchtools.PitchRange(range_string='[A0, C8]')
```

A pitch range.

```
>>> pitch_range = pitchtools.PitchRange(' [C3, C7]')
>>> print(format(pitch_range))
pitchtools.PitchRange(
    range_string=' [C3, C7]',
)
```

```
>>> 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`
- `abctools.AbjadObject.AbstractBase`
- `__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.range_string`
Gets range string of pitch range.

```
>>> pitch_range.range_string
'[C3, C7]'
```

Aliased to *one_line_named_pitch_repr*.

Returns string.

`PitchRange.start_pitch`
Start pitch of pitch range.

```
>>> pitch_range.start_pitch
NamedPitch('c')
```

Returns pitch.

`PitchRange.start_pitch_is_included_in_range`
Is 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`

Is true when stop pitch is included in range. Otherwise false:

```
>>> pitch_range.stop_pitch_is_included_in_range
True
```

Returns boolean.

Methods

`PitchRange.voice_pitch_class` (*pitch_class*)

Voices *pitch_class* in this pitch-range.

```
>>> a_pitch_range = pitchtools.PitchRange(' [C4, C6] ')
>>> a_pitch_range.voice_pitch_class('c')
(NamedPitch("c'"), NamedPitch("c''"), NamedPitch("c'''"))
```

```
>>> a_pitch_range.voice_pitch_class('b')
(NamedPitch("b'"), NamedPitch("b''"))
```

```
>>> a_pitch_range = pitchtools.PitchRange(' [C4, A4] ')
>>> a_pitch_range.voice_pitch_class('b')
()
```

Returns tuple of zero or more named pitches.

Class methods

`PitchRange.is_range_string` (*expr*)

Is true when *expr* is a symbolic pitch range string. Otherwise false:

```
>>> pitchtools.PitchRange.is_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.

Static methods

`PitchRange.from_pitches` (*start_pitch*, *stop_pitch*, *start_pitch_is_included_in_range=True*,
stop_pitch_is_included_in_range=True)

Initializes pitch range from numbers.

```
>>> pitchtools.PitchRange.from_pitches(-18, 19)
PitchRange(range_string=' [F#2, G5] ')
```

Returns pitch range.

Special methods

`PitchRange.__contains__` (*arg*)

Is true when pitch range contains *arg*. Otherwise false.

Returns boolean.

`PitchRange.__eq__` (*expr*)

Is true when *expr* 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)`

Is true when start pitch of pitch range is greater than or equal to *arg*. Otherwise false.

Returns boolean.

`PitchRange.__gt__(arg)`

Is true when start pitch of pitch range is greater than *arg*. Otherwise false.

Returns boolean.

`PitchRange.__hash__()`

Hashes pitch range.

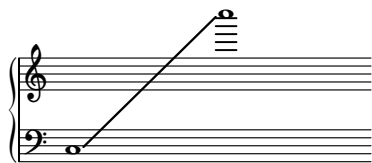
Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`PitchRange.__illustrate__()`

Illustrates pitch range.

```
>>> show(pitch_range)
```



Returns LilyPond file.

`PitchRange.__le__(arg)`

Is true when stop pitch of pitch-range is less than or equal to *arg*. Otherwise false.

Returns boolean.

`PitchRange.__lt__(arg)`

Is true when stop pitch of pitch-range is less than *arg*. Otherwise false.

Returns boolean.

`PitchRange.__ne__(arg)`

Is true when pitch range does not equal *arg*. Otherwise false.

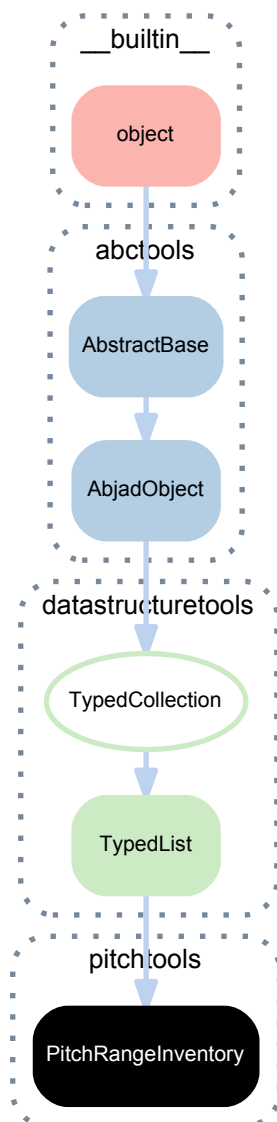
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

12.2.33 pitchtools.PitchRangeInventory



class `pitchtools.PitchRangeInventory` (*items=None, item_class=None, keep_sorted=None*)
 An ordered list of pitch ranges.

```

>>> inventory = pitchtools.PitchRangeInventory([
...     '[C3, C6]',
...     '[C4, C6]',
...     ])
>>> inventory
PitchRangeInventory([PitchRange(range_string='[C3, C6]'), PitchRange(range_string='[C4, C6]')])
  
```

Pitch range inventories implement list interface and are mutable.

Bases

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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(item)`
Changes *item* to item and returns index.

```
>>> pitch_collection = datastructuretools.TypedList(
...     items=('cqf', "as'", 'b', 'dss'),
...     item_class=NamedPitch)
>>> pitch_collection.index("as'")
1
```

Returns index.

(TypedList) .**insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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__** (*item*)

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

(TypedList) .**__delitem__** (*i*)

Aliases list.__delitem__().

Returns none.

(TypedCollection) .**__eq__** (*expr*)

Is 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__**().

Returns item.

(TypedCollection).**__hash__**()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedList).**__iadd__**(*expr*)

Changes items 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(
            name='ppp',
        ),
        indicatortools.Dynamic(
            name='p',
        ),
        indicatortools.Dynamic(
            name='mp',
        ),
        indicatortools.Dynamic(
            name='mf',
        ),
        indicatortools.Dynamic(
            name='fff',
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

PitchRangeInventory.**__illustrate__**()

Illustrates pitch range inventory.

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



Returns LilyPond file.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__**().
 Returns generator.

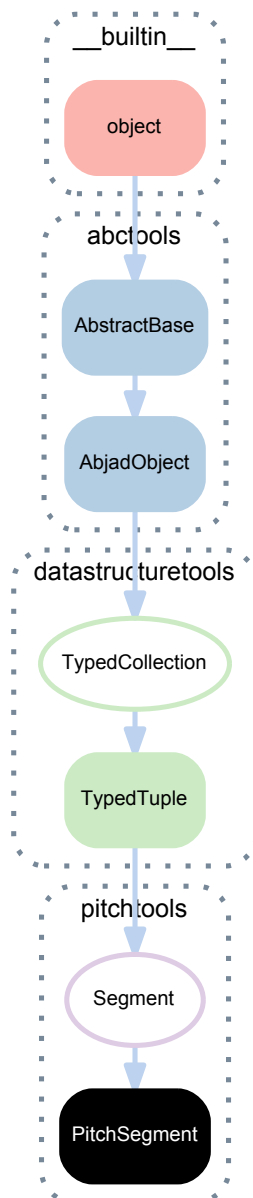
(TypedList).**__setitem__**(*i, expr*)
 Changes items in *expr* to items and sets.

```
>>> 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,
)
```

Returns none.

12.2.34 pitchtools.PitchSegment



class `pitchtools.PitchSegment` (*items*=(-2, -1.5, 6, 7, -1.5, 7), *item_class*=None)

A pitch segment.

```
>>> numbered_pitch_segment = pitchtools.PitchSegment(
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     item_class=pitchtools.NumberedPitch,
... )
>>> numbered_pitch_segment
PitchSegment([-2, -1.5, 6, 7, -1.5, 7])
```

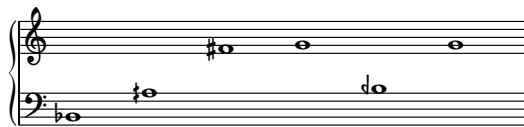
```
>>> show(numbered_pitch_segment)
```



```
>>> named_pitch_segment = pitchtools.PitchSegment(
...     ['bf', 'aqs', 'fs', 'g', 'bqf', 'g'],
...     item_class=NamedPitch,
```

```
...    )
>>> named_pitch_segment
PitchSegment(['bf', 'aq', 'fs', 'g', 'bqf', 'g'])
```

```
>>> show(named_pitch_segment)
```



Pitch segments are immutable.

Bases

- `pitchtools.Segment`
- `datastructuretools.TypedTuple`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`PitchSegment.has_duplicates`

True if pitch segment has duplicate items. Otherwise false.

```
>>> pitch_class_segment = pitchtools.PitchSegment(
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.has_duplicates
True
```

```
>>> pitch_class_segment = pitchtools.PitchSegment(
...     items="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 items into.

`(TypedCollection).items`

Gets collection items.

`PitchSegment.local_maxima`

Local maxima of pitch segment.

Returns tuple.

`PitchSegment.local_minima`

Local minima of pitch segment.

Returns tuple.

Methods

(`TypedTuple`) **.count** (*item*)
Changes *item* to item.

Returns count in collection.

(`TypedTuple`) **.index** (*item*)
Changes *item* 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)
```



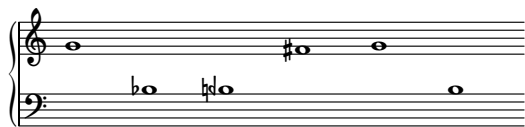
Returns new pitch segment.

`PitchSegment.rotate(n, transpose=False)`

Rotates pitch segment.

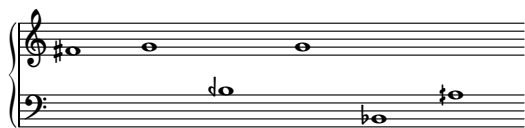
```
>>> result = numbered_pitch_segment.rotate(1)
>>> result
PitchSegment([7, -2, -1.5, 6, 7, -1.5])
```

```
>>> show(result)
```



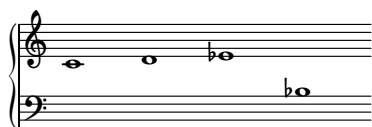
```
>>> result = named_pitch_segment.rotate(-2)
>>> result
PitchSegment(['fs', 'g', 'bqf', 'g', 'bf', 'aqs'])
```

```
>>> show(result)
```



```
>>> pitch_segment = pitchtools.PitchSegment("c' d' e' f'")
>>> result = pitch_segment.rotate(-1, transpose=True)
>>> result
PitchSegment(['c', 'd', 'ef', 'bf'])
```

```
>>> 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)`

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

`(TypedCollection) .__eq__ (expr)`
Is 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.

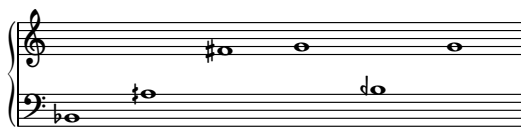
`(TypedTuple) .__getitem__ (i)`
Gets *i* from type tuple.
Returns item.

`(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', 'ags', "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.

`(TypedTuple) .__mul__ (expr)`
Multiplies typed tuple by *expr*.
Returns new typed tuple.

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

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

Returns boolean.

(TypedTuple) .**__radd__**(*expr*)

Right-adds *expr* to typed tuple.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

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

Multiplies *expr* by typed tuple.

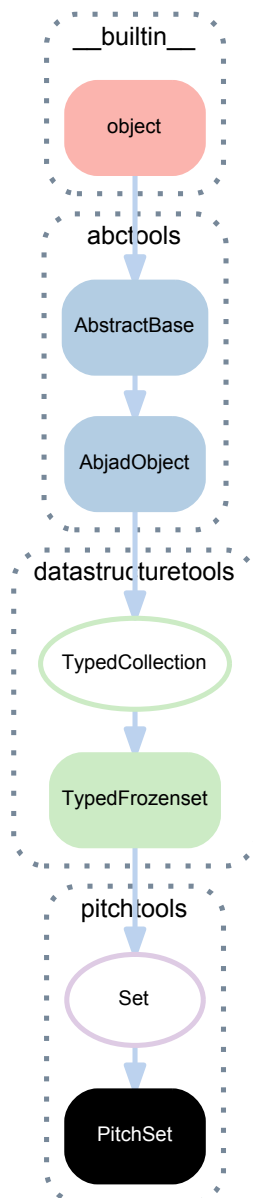
Returns new typed tuple.

(Segment) .**__str__**()

String representation of segment.

Returns string.

12.2.35 pitchtools.PitchSet



class `pitchtools.PitchSet` (*items=None, item_class=None*)
 A pitch set.

```
>>> numbered_pitch_set = pitchtools.PitchSet(
...     items=[-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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`PitchSet`.**`duplicate_pitch_classes`**

Duplicate pitch-classes in pitch set.

Returns pitch-class set.

`PitchSet`.**`is_pitch_class_unique`**

Is true when pitch set is pitch-class-unique. Otherwise false.

Returns boolean.

`(TypedCollection)`.**`item_class`**

Item class to coerce items into.

`(TypedCollection)`.**`items`**

Gets collection items.

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*)

Is true when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet)`.**`issubset`** (*expr*)

Is true when typed frozen set is a subset of *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet)`.**`issuperset`** (*expr*)

Is 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)`
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__(item)`
Is true when typed collection contains *item*. Otherwise false.

Returns boolean.

`(TypedCollection).__eq__(expr)`
Is 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)`
Is true when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet).__gt__(expr)`
Is 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*)
Is 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*)
Is true when typed frozen set is less than *expr*. Otherwise false.
Returns boolean.

(`TypedFrozenSet`) .**__ne__**(*expr*)
Is 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.

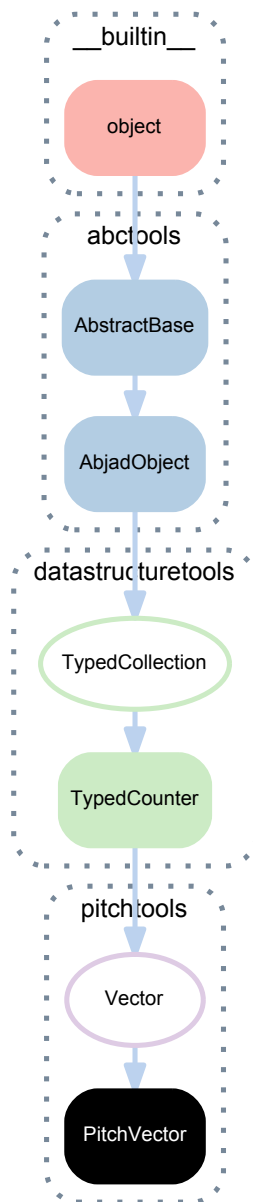
(`AbjadObject`) .**__repr__**()
Gets interpreter representation of Abjad object.
Returns string.

(`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.

12.2.36 pitchtools.PitchVector



class `pitchtools.PitchVector` (*items=None, item_class=None*)
 A pitch vector.

Bases

- `pitchtools.Vector`
- `datastructuretools.TypedCounter`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

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

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()`
Iterates items in typed counter.

Yields items.

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

`(TypedCounter).viewitems()`
Please document.

`(TypedCounter).viewkeys()`
Please document.

`(TypedCounter).viewvalues()`
Please document.

Class methods

`PitchVector.from_selection(selection, item_class=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__** (*item*)
Is true when typed collection container *item*. Otherwise false.
Returns boolean.

(TypedCounter) .**__delitem__** (*item*)
Deletes *item* from typed counter.
Returns none.

(TypedCollection) .**__eq__** (*expr*)
Is 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__** (*item*)
Gets *item* from typed counter.
Returns item.

(TypedCollection) .**__hash__** ()
Hashes typed collection.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
Returns integer.

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

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

(TypedCounter) .**__missing__** (*item*)
Returns zero.
Returns zero.

(TypedCollection) .**__ne__** (*expr*)
Is 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__**(*item*, *value*)

Sets typed counter *item* to *value*.

Returns none.

(Vector).**__str__**()

String representation of vector.

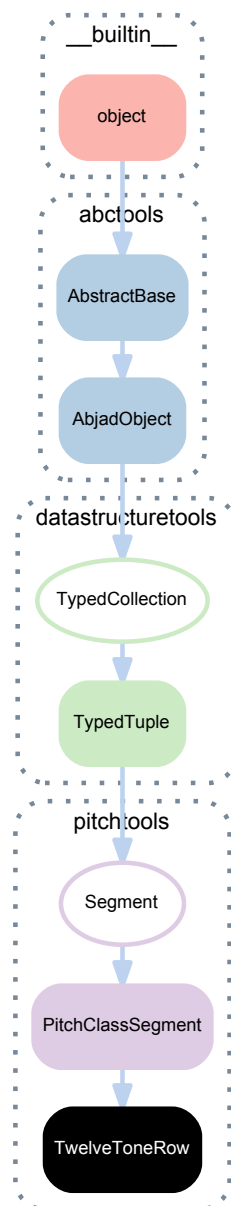
Returns string.

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

Subtracts *expr* from typed counter.

Returns new typed counter.

12.2.37 pitchtools.TwelveToneRow



class pitchtools.**TwelveToneRow**(*items*=(0, 1, 11, 9, 3, 6, 7, 5, 4, 10, 2, 8))

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

Read-only properties

`(PitchClassSegment).has_duplicates`

True if segment contains duplicate items:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     items=[-2, -1.5, 6, 7, -1.5, 7],
... )
>>> pitch_class_segment.has_duplicates
True
```

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     items="c d e f g a b",
... )
>>> pitch_class_segment.has_duplicates
False
```

Returns boolean.

`(TypedCollection).item_class`

Item class to coerce items into.

`(TypedCollection).items`

Gets collection items.

Methods

`(PitchClassSegment).alpha()`

Morris alpha transform of pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     items=[-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** (*item*)

Changes *item* to item.

Returns count in collection.

(TypedTuple) **.index** (*item*)

Changes *item* to item.

Returns index in collection.

(PitchClassSegment) **.invert** ()

Invert pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     items=[-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 (
...     items=[-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 (
...     items=[-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*, *transpose=False*)

Rotate pitch-class segment:

```
>>> pitchtools.PitchClassSegment (
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     ).rotate(1)
PitchClassSegment([7, 10, 10.5, 6, 7, 10.5])
```

```
>>> pitchtools.PitchClassSegment (
...     items=['c', 'ef', 'bqs', 'd'],
...     ).rotate(-2)
PitchClassSegment(['bqs', 'd', 'c', 'ef'])
```

If *transpose* is true, transpose the rotated segment to begin at the same pitch class as this segment:

```
>>> pitchtools.PitchClassSegment (
...     items=['c', 'b', 'd']
...     ).rotate(1, transpose=True)
PitchClassSegment(['c', 'bf', 'a'])
```

Returns new pitch-class segment.

(PitchClassSegment) **.transpose** (*expr*)

Transpose pitch-class segment:

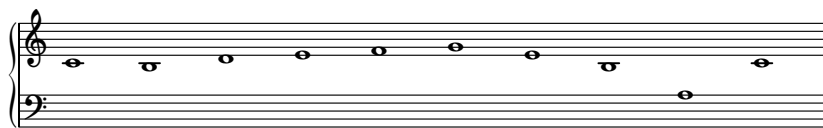
```
>>> pitchtools.PitchClassSegment (
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     ).transpose(10)
PitchClassSegment([8, 8.5, 4, 5, 8.5, 5])
```

Returns new pitch-class segment.

(PitchClassSegment) **.voice_horizontally** (*initial_octave=4*)

Voices pitch-class segment as pitch segment, with each pitch as close in distance to the previous pitch as possible.

```
>>> pitch_classes = pitchtools.PitchClassSegment (
...     "c b d e f g e b a c")
>>> pitch_segment = pitch_classes.voice_horizontally()
>>> show(pitch_segment)
```

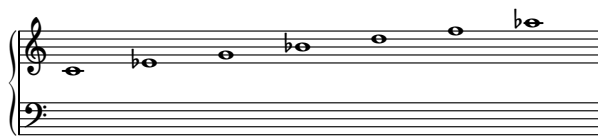


Returns pitch segment.

(PitchClassSegment) **.voice_vertically** (*initial_octave=4*)

Voices pitch-class segment as pitch segment, with each pitch always higher than the previous.

```
>>> scale_degree_numbers = [1, 3, 5, 7, 9, 11, 13]
>>> scale = tonalanalysistools.Scale('c', 'minor')
>>> pitch_classes = pitchtools.PitchClassSegment((
...     scale.scale_degree_to_named_pitch_class(x)
...     for x in scale_degree_numbers))
>>> pitch_segment = pitch_classes.voice_vertically()
>>> pitch_segment
PitchSegment(['c', 'ef', 'g', 'bf', 'd', 'f', 'af'])
>>> show(pitch_segment)
```



Returns pitch segment.

Class methods

`TwelveToneRow.from_selection(selection, item_class=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__(item)`

Change *item* to item and return true if item exists in collection.

Returns none.

`(TypedCollection).__eq__(expr)`

Is 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.__mul__(expr)`
 Multiplies twelve-tone row by *expr*.
 Returns pitch-class segment.

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

`(TypedTuple).__radd__(expr)`
 Right-adds *expr* to typed tuple.

`(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.

12.3 Functions

12.3.1 `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.

12.3.2 `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.

12.3.3 `pitchtools.contains_subsegment`

`pitchtools.contains_subsegment` (*pitch_class_numbers*, *pitch_numbers*)

Is 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.

12.3.4 `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.

12.3.5 `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.

12.3.6 `pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list`

`pitchtools.insert_and_transpose_nested_subruns_in_pitch_class_number_list` (*notes*, *subrun_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.

12.3.7 `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.

12.3.8 `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.

12.3.9 `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):
...     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.

12.3.10 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.

12.3.11 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.

12.3.12 `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.

12.3.13 `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.from_pitches(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.

12.3.14 `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.

12.3.15 `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.

12.3.16 `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.

12.3.17 `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.

12.3.18 `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.

12.3.19 `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.

12.3.20 `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.

12.3.21 `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.

12.3.22 `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.

12.3.23 `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.

12.3.24 `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.

12.3.25 `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.

12.3.26 `pitchtools.suggest_clef_for_named_pitches`

`pitchtools.suggest_clef_for_named_pitches` (*pitches*)

Suggest clef for named *pitches*:

```
>>> staff = Staff(scoretools.make_notes(
...     list(range(-12, -6)), [(1, 4)]))
>>> pitchtools.suggest_clef_for_named_pitches(staff)
Clef(name='bass')
```

Suggest clef based on minimal number of ledger lines.

Returns clef.

12.3.27 `pitchtools.transpose_named_pitch_by_numbered_interval_and_respell`

`pitchtools.transpose_named_pitch_by_numbered_interval_and_respell` (*pitch*,
staff_spaces,
numbered_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.

12.3.28 `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*.

12.3.29 `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.

12.3.30 `pitchtools.transpose_pitch_expr_into_pitch_range`

`pitchtools.transpose_pitch_expr_into_pitch_range` (*pitch_expr*, *pitch_range*)
Transpose *pitch_expr* into *pitch_range*:

```
>>> pitches = [-2, -1, 13, 14]  
>>> range_ = pitchtools.PitchRange('C4, C5')  
>>> pitchtools.transpose_pitch_expr_into_pitch_range(pitches, range_)  
[10, 11, 1, 2]
```

Returns new *pitch_expr* object.

12.3.31 `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*:

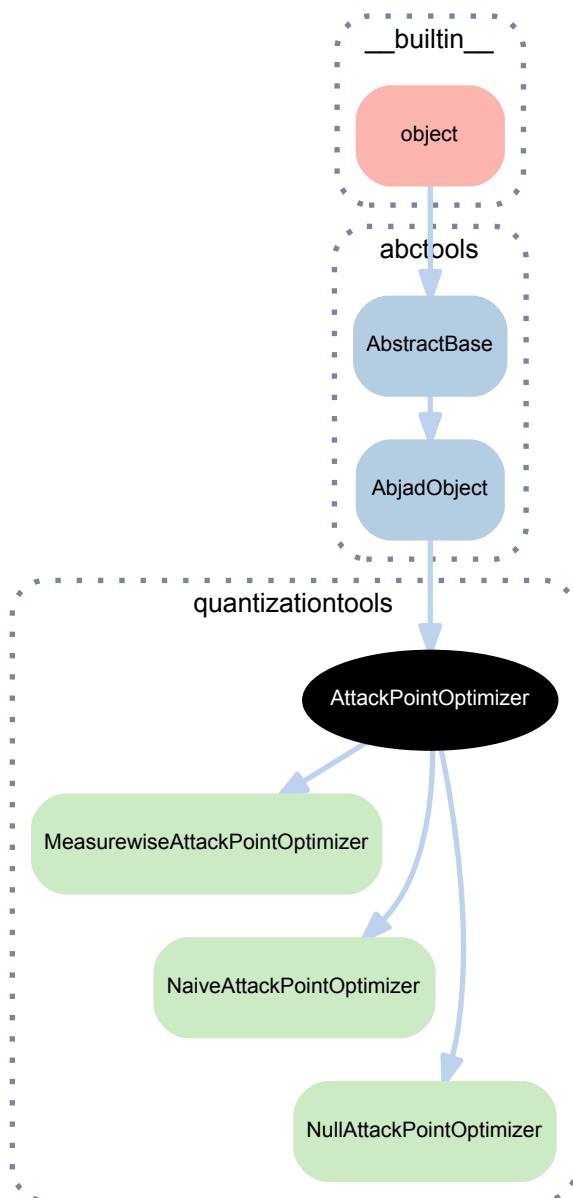
```
>>> pitch_numbers = [-30, -18, -6, 6, 18, 30, 42]
>>> for n in pitch_numbers:
...     n, pitchtools.transpose_pitch_number_by_octave_transposition_mapping(
...         n, mapping)
...
(-30, 6)
(-18, 6)
(-6, 18)
(6, 18)
(18, 18)
(30, 30)
(42, 30)
```

And so on.

Returns pitch number.

13.1 Abstract classes

13.1.1 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to ‘’ or ‘*storage*’. Interprets ‘’ equal to ‘*storage*’.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

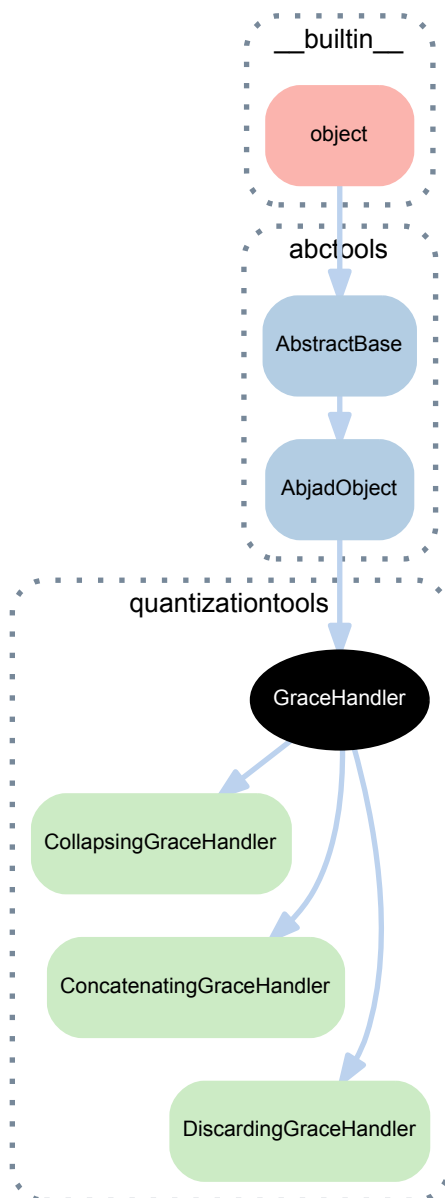
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.1.2 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

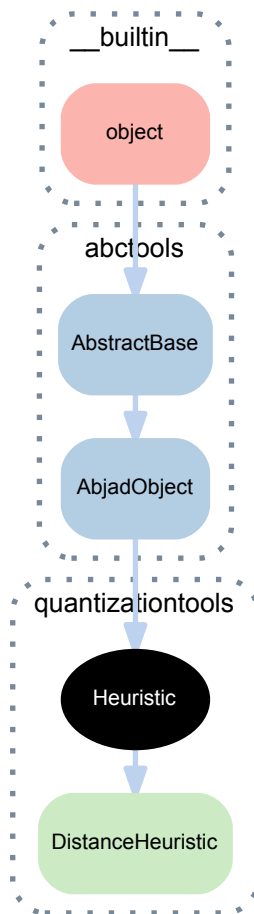
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.1.3 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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

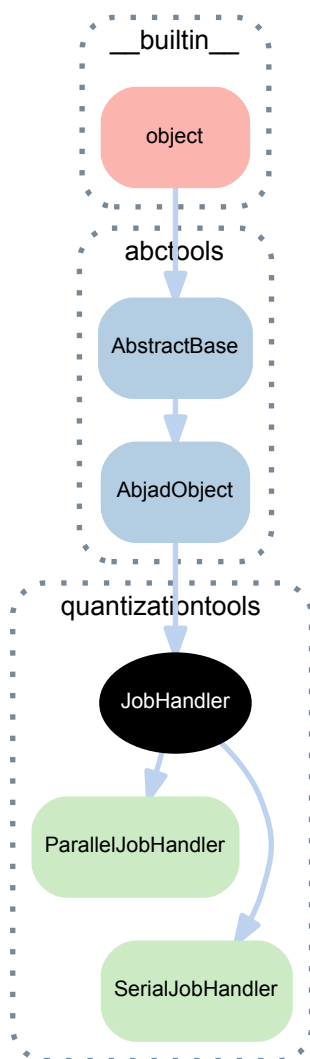
(AbjadObject).**__format__**(*format_specification*='')
 Formats Abjad object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject).**__hash__**()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 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 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

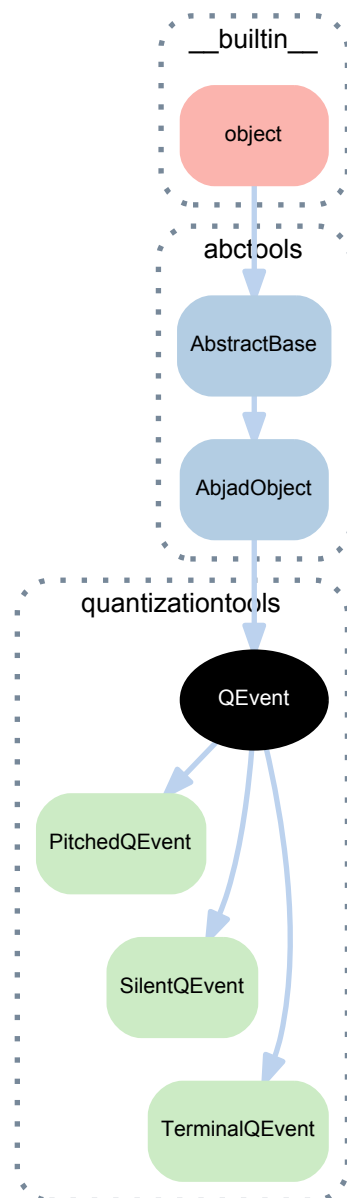
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.1.5 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`QEvent.__lt__(expr)`

Is 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.

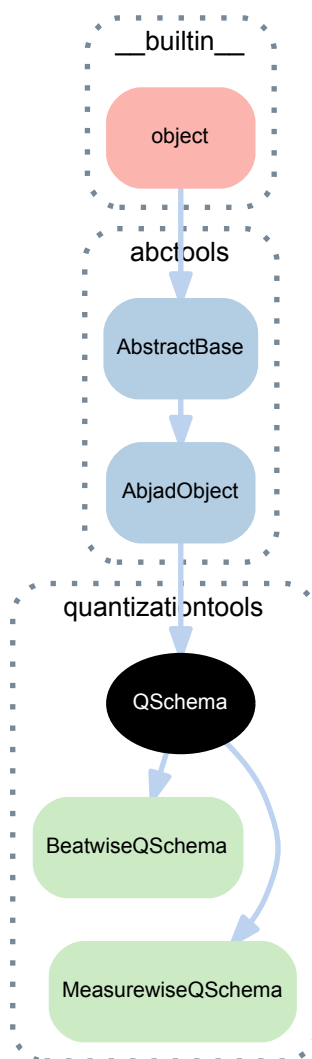
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.1.6 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`
- `abctools.AbjadObject.AbstractBase`
- `__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).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

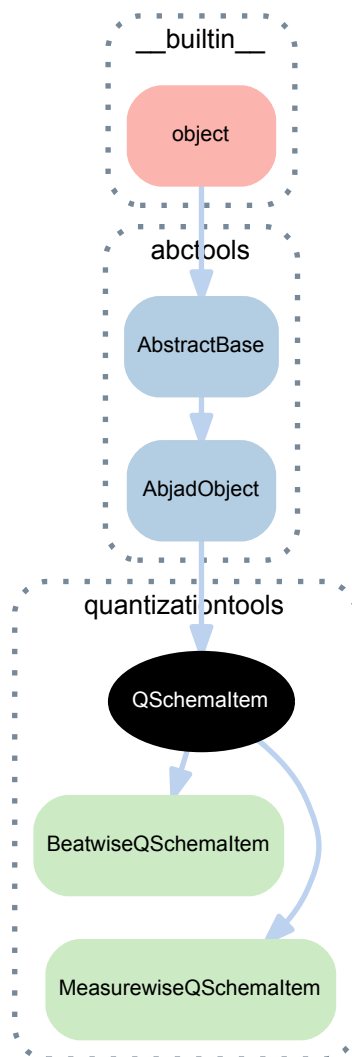
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.1.7 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`
- `abctools.AbjadObject.AbstractBase`
- `__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).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

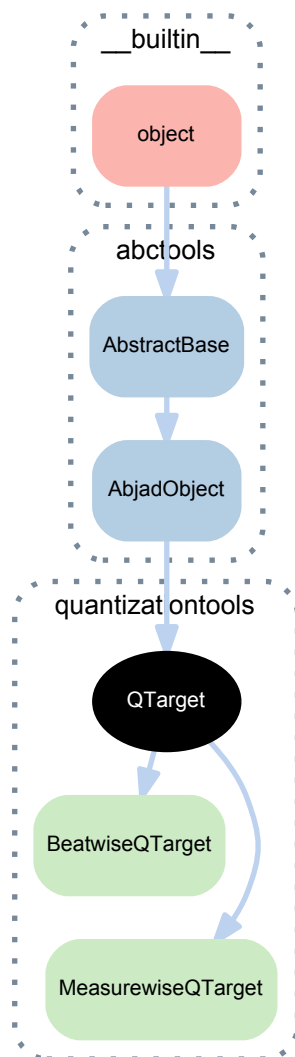
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.1.8 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(`AbjadObject`).`__hash__`()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

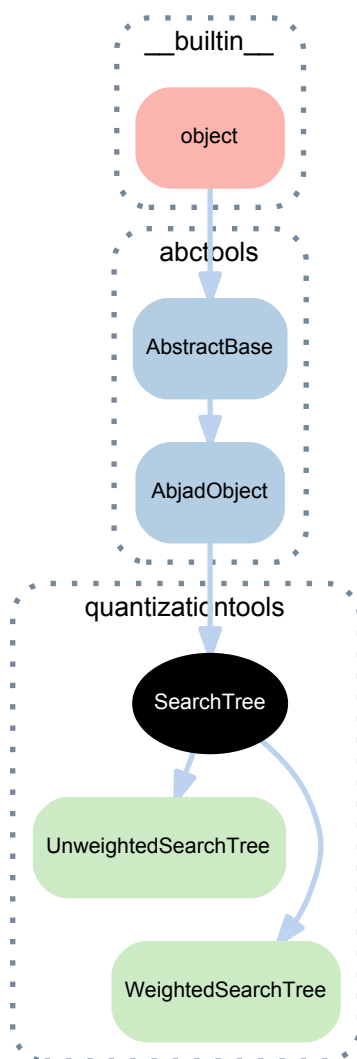
Returns boolean.

(`AbjadObject`).`__repr__`()

Gets interpreter representation of Abjad object.

Returns string.

13.1.9 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`SearchTree.__hash__()`

Hashes search tree.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

Returns boolean.

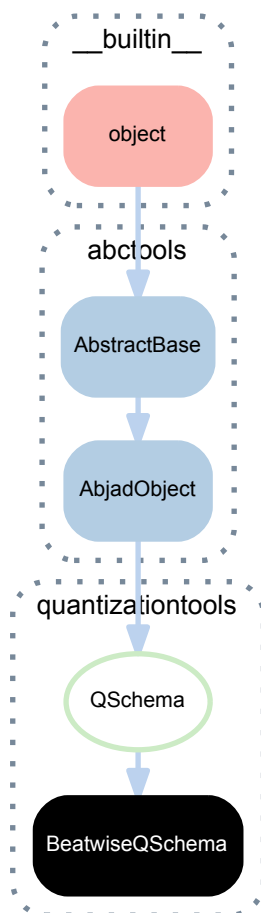
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2 Concrete classes

13.2.1 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: 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(
        duration=durationtools.Duration(1, 4),
        units_per_minute=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: {
          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`
- `abctools.AbjadObject.AbstractBase`
- `__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).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

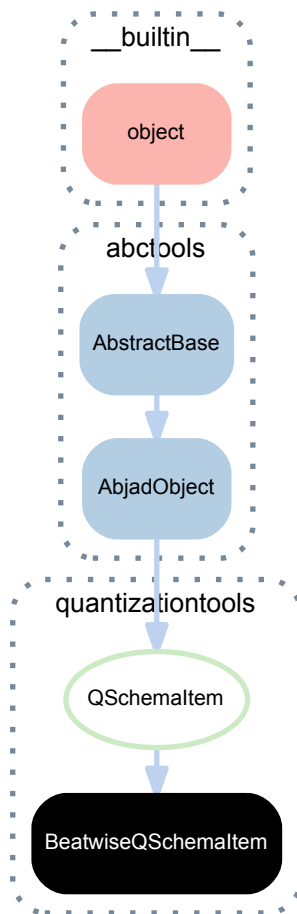
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.2.2 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(
        duration=durationtools.Duration(1, 4),
        units_per_minute=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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

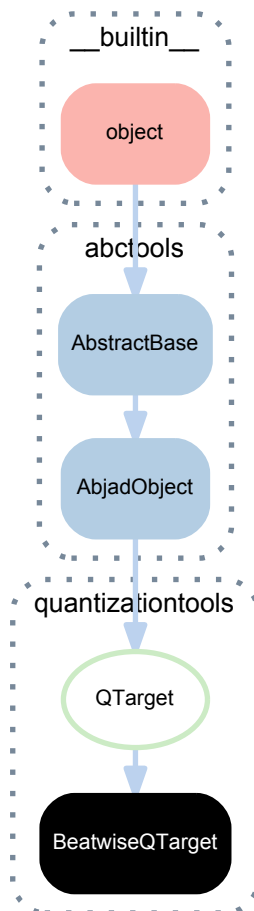
`(QSchemaItem).__format__(format_specification='')`
Formats q schema item.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

13.2.3 quantizationtools.BeatwiseQTarget



class `quantizationtools.BeatwiseQTarget` (*items=None*)

A beat-wise quantization target.

Not composer-safe.

Used internally by `Quantizer`.

Bases

- `quantizationtools.QTarget`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

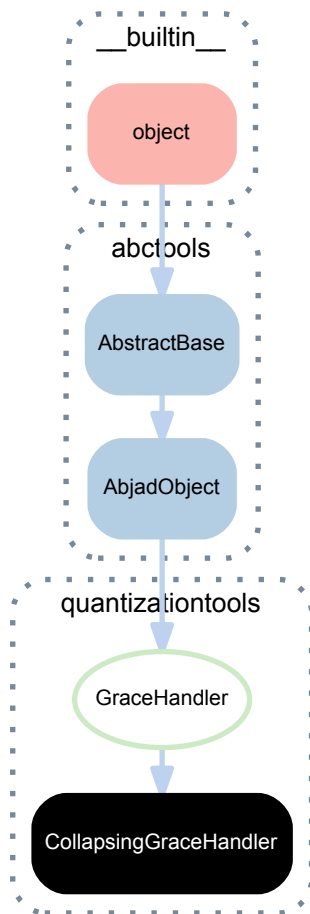
`(AbjadObject).__format__(format_specification='')`
Formats Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

13.2.4 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

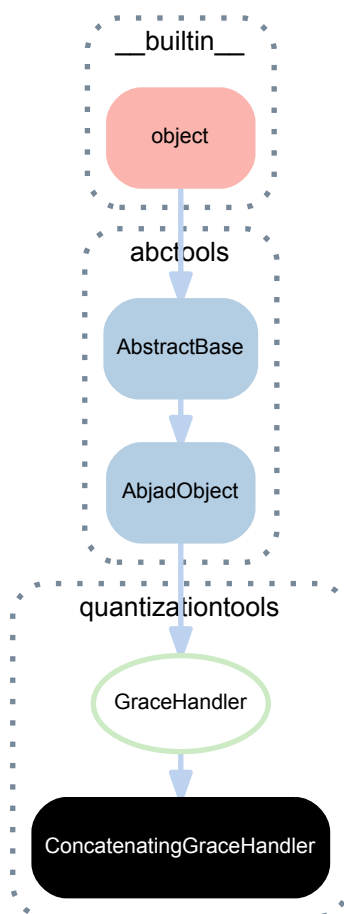
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.2.5 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`
- `abctools.AbjadObject.AbstractBase`

- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

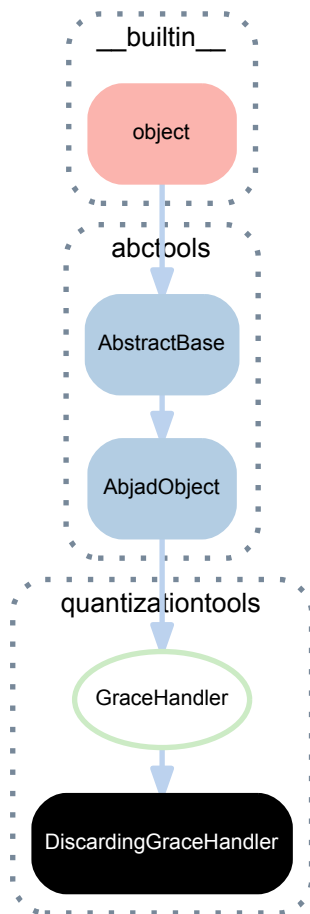
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2.6 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

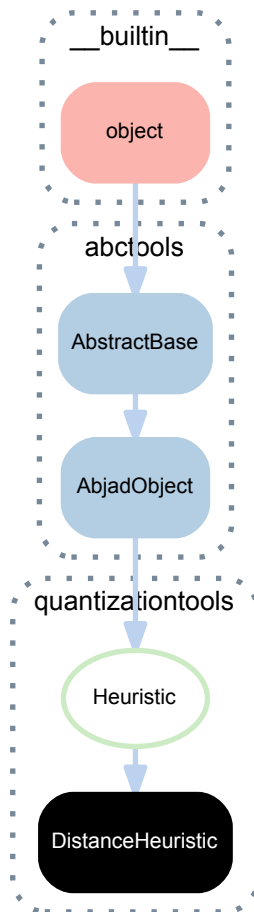
(AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

Returns string.

13.2.7 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

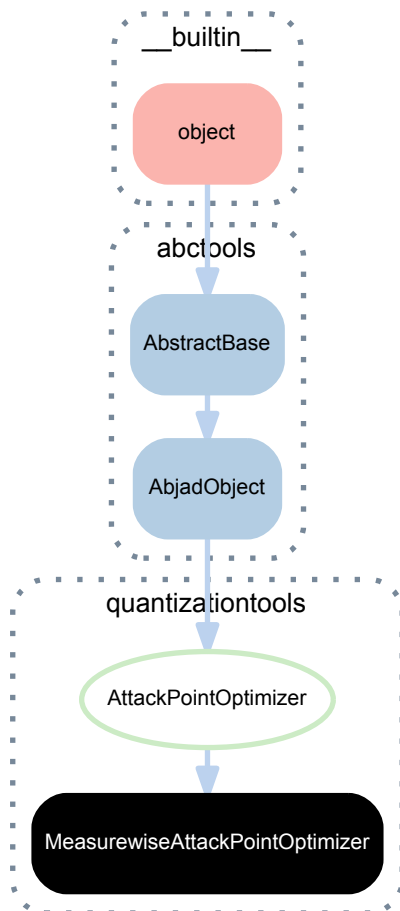
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2.8 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.

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



```
>>> source_tempo = Tempo((1, 4), 60)
>>> q_events = quantizationtools.QEventSequence.from_tempo_scaled_leaves(
...     staff.select_leaves(),
...     tempo=source_tempo,
... )
>>> target_tempo = Tempo((1, 4), 54)
>>> q_schema = quantizationtools.MeasurewiseQSchema(
...     tempo=target_tempo,
... )
>>> quantizer = quantizationtools.Quantizer()
```

Without the measure-wise attack-point optimizer:

```
>>> result = quantizer(
...     q_events,
...     q_schema=q_schema,
... )
>>> show(result)
```



With the measure-wise attack-point optimizer:

```
>>> optimizer = quantizationtools.MeasurewiseAttackPointOptimizer()
>>> result = quantizer(
...     q_events,
...     attack_point_optimizer=optimizer,
...     q_schema=q_schema,
... )
>>> show(result)
```



Only acts on Measure instances.

Bases

- `quantizationtools.AttackPointOptimizer`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

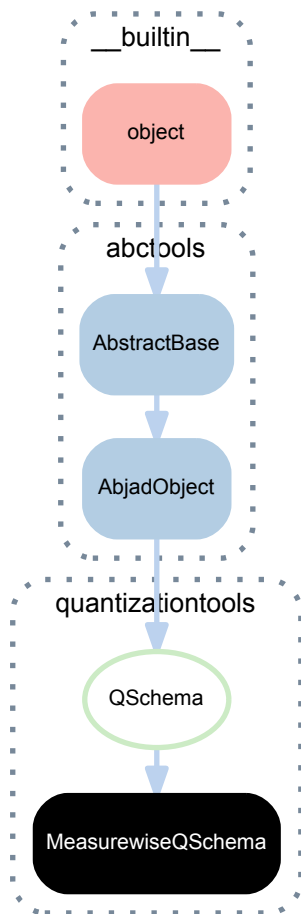
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2.9 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(
    duration=durationtools.Duration(1, 4),
    units_per_minute=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`
- `abctools.AbjadObject.AbstractBase`
- `__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).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

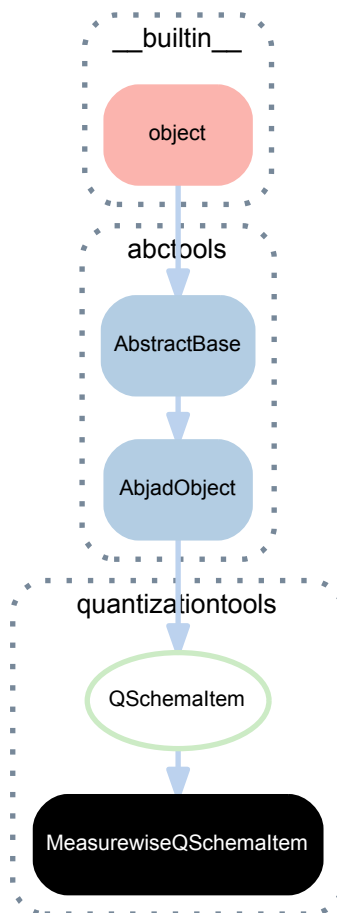
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.2.10 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(
    duration=durationtools.Duration(1, 4),
    units_per_minute=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`
- `abctools.AbjadObject.AbstractBase`
- `__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).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

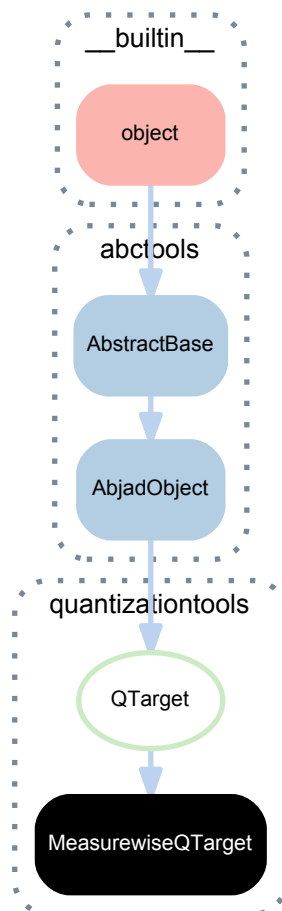
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.2.11 quantizationtools.MeasurewiseQTarget



class `quantizationtools.MeasurewiseQTarget` (*items=None*)
A measure-wise quantization target.
Not composer-safe.
Used internally by `Quantizer`.

Bases

- `quantizationtools.QTarget`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

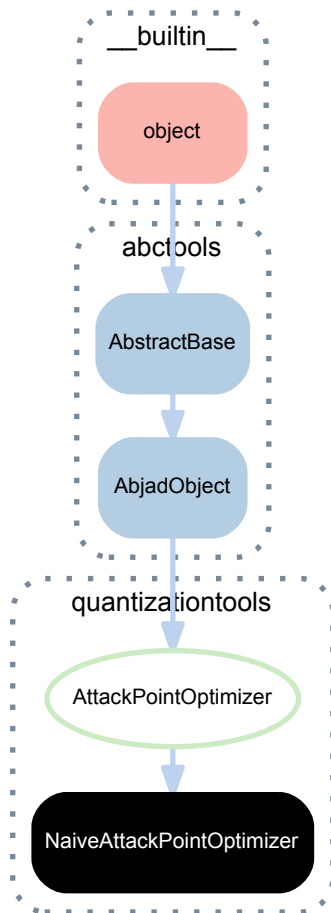
`(AbjadObject).__format__(format_specification='')`
Formats Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

13.2.12 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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

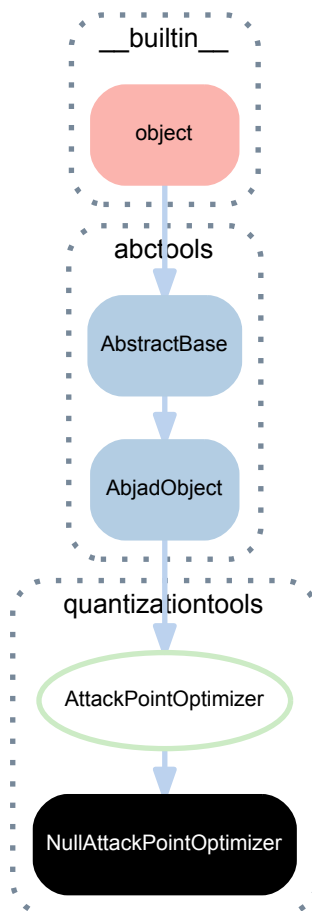
(AbjadObject) .**__format__**(*format_specification*='')
 Formats Abjad object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject) .**__hash__**()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 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.

13.2.13 quantizationtools.NullAttackPointOptimizer



class quantizationtools.**NullAttackPointOptimizer**
 Concrete AttackPointOptimizer subclass which performs no attack point optimization.

Bases

- `quantizationtools.AttackPointOptimizer`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

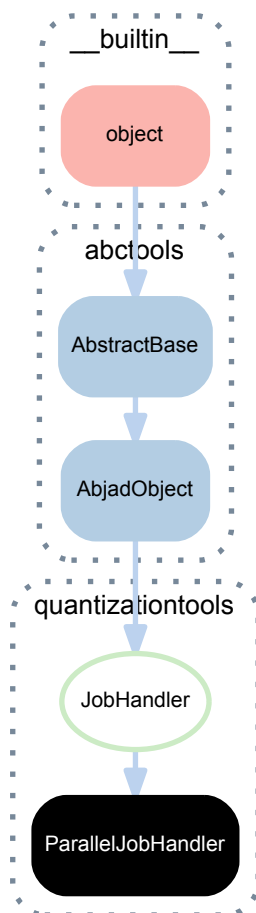
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2.14 quantizationtools.ParallelJobHandler



class `quantizationtools.ParallelJobHandler`

Processes `QuantizationJob` instances in parallel, based on the number of CPUs available.

Bases

- `quantizationtools.JobHandler`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

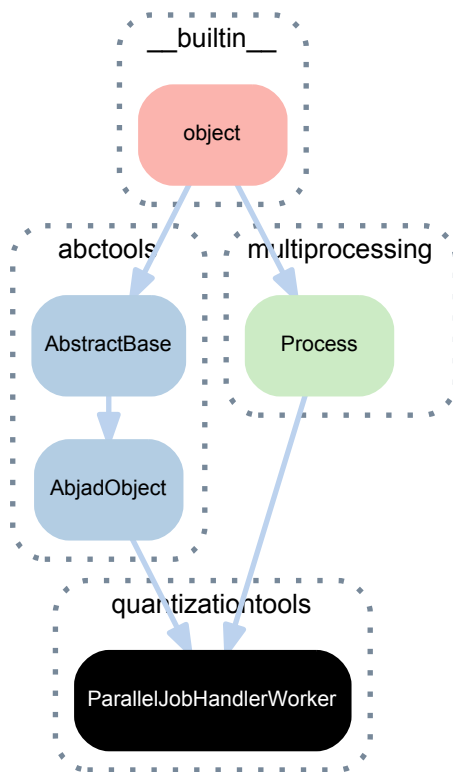
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.2.15 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.
- (AbjadObject) **.__hash__()**
Hashes Abjad object.

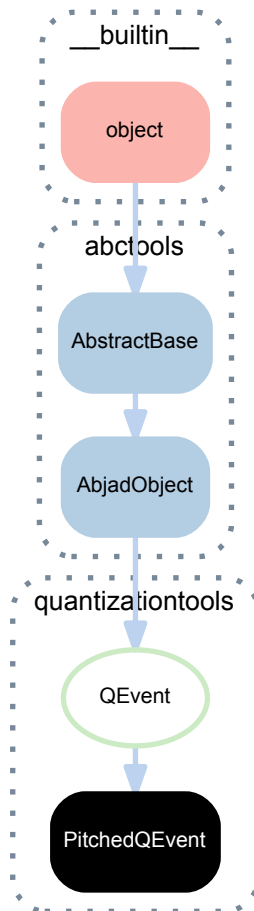
Required to be explicitly re-defined on Python 3 if **__eq__** changes.

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

Returns boolean.


```
(Process).__repr__()
```

13.2.16 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`PitchedQEvent.__hash__()`

Hashes pitched q-event.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(QEvent).__lt__(expr)`

Is 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.

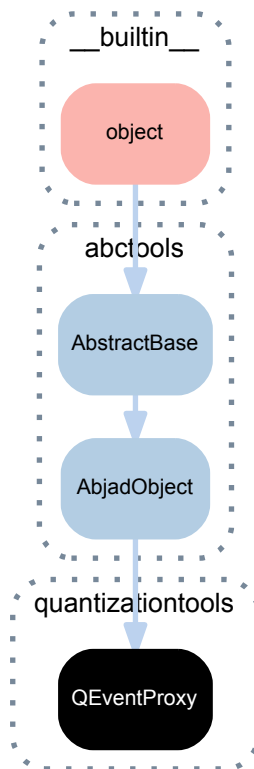
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2.17 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(
  quantizationtools.PitchedQEvent(
    offset=durationtools.Offset(130, 1),
    pitches=(
      pitchtools.NamedPitch("c'"),
      pitchtools.NamedPitch("cs'"),
      pitchtools.NamedPitch("e'"),
    ),
  ),
  durationtools.Offset(1, 2)
)

```

Not composer-safe.

Used internally by Quantizer.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)`
Is 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.

`QEventProxy.__hash__()`
Hashes q-event proxy.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

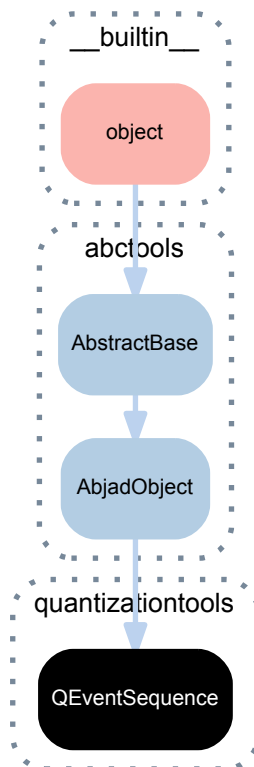
`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

Returns string.

13.2.18 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`
- `abctools.AbjadObject.AbstractBase`
- `__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(millisecons,fuse_silences=False)`

Convert a sequence of millisecond 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 = tuple(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)`

Is true when q-event sequence contains *expr*. Otherwise false.

Returns boolean.

`QEventSequence.__eq__(expr)`

Is 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.__hash__()`

Hashes q-event sequence.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`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.

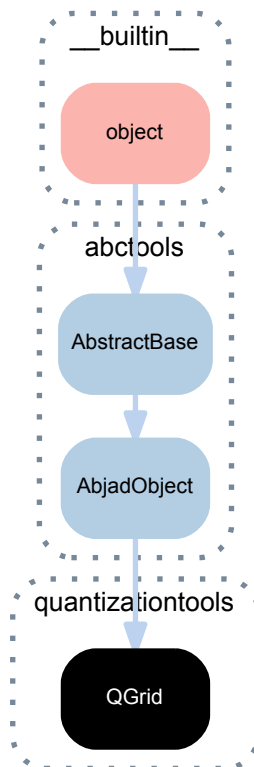
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.2.19 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(
    quantizationtools.PitchedQEvent(
        offset=durationtools.Offset(250, 1),
        pitches=(
            pitchtools.NamedPitch("c'"),
        ),
    ),
    durationtools.Offset(1, 4)
)
```

```
>>> for q_event_proxy in q_grid.next_downbeat.q_event_proxies:
...     print(format(q_event_proxy, 'storage'))
...
quantizationtools.QEventProxy(
    quantizationtools.PitchedQEvent(
        offset=durationtools.Offset(750, 1),
        pitches=(
            pitchtools.NamedPitch("cs'"),
        ),
    ),
    durationtools.Offset(3, 4)
)
```

Used internally by the Quantizer.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.__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.

`QGrid.__hash__()`

Hashes `q-grid`.

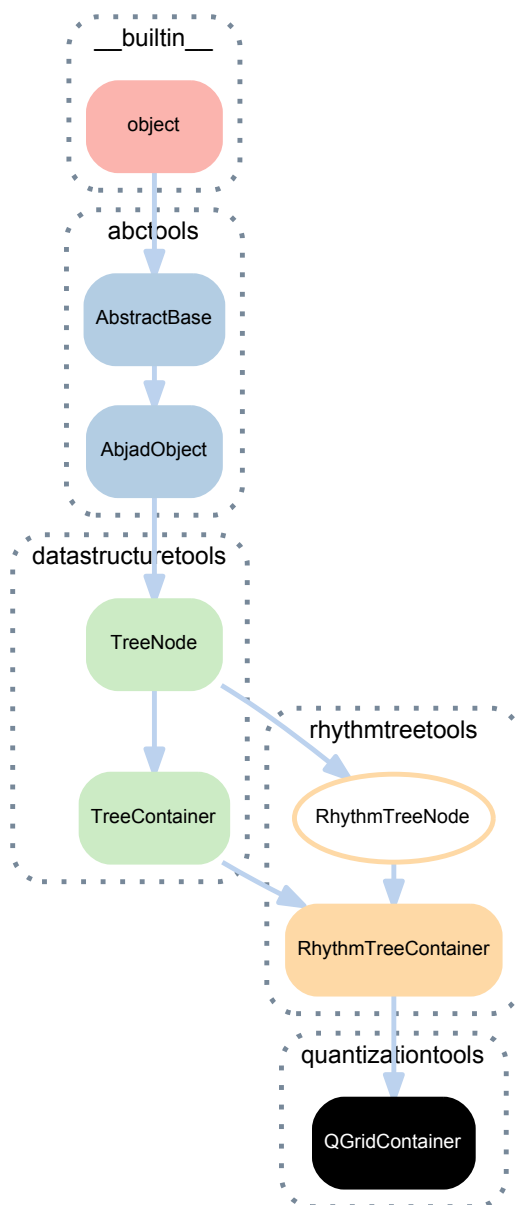
Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

13.2.20 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

- `rhythmtreertools.RhythmTreeContainer`
- `rhythmtreertools.RhythmTreeNode`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 = rhythmtreertools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

(TreeNode).**graph_order**

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 = rhythmtreertools.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];
```

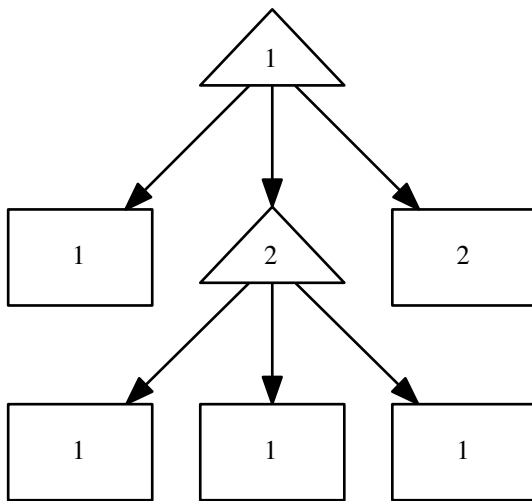


```

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 = 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.

(RhythmTreeContainer) **.rtm_format**

The node's RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreertools.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 = 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

(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 = rhythmtreertools.RhythmTreeParser() ('(1 (1 1 1))')[0]
>>> b = rhythmtreertools.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 FixedDurationTuplet(Duration(1, 8), "c\'16 c\'16 c\'16") c\'16)

```

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 Abjad 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.

(RhythmTreeContainer).**__hash__**()

Hashes rhythm-tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

Is 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
```

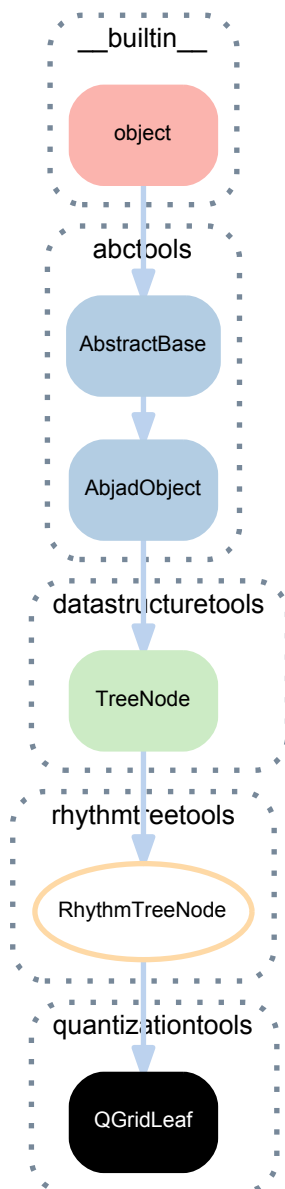
```
>>> c.parent is a
True
```

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

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

Return *None*.

13.2.21 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

- `rhythmtreetools.RhythmTreeNode`
- `datastructuretools.TreeNode`

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 = rhythmtreertools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

(TreeNode) **.graph_order**
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 = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)

>>> a.extend([b, c])
>>> b.extend([d, e])

>>> a.parentage_ratios
(Duration(1, 1),)

>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))

>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))

>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))

>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Returns tuple.

`QGridLeaf`.**`preceding_q_event_proxies`**

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 = rhythmtreetools.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.

(TreeNode) .**__deepcopy__** (*args)

Copies tree node.

Returns new tree node.

QGridLeaf .**__eq__** (*expr*)

Is 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 Abjad object.

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

Returns string.

QGridLeaf .**__hash__** ()

Hashes q-grid leaf.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TreeNode) .**__ne__** (*expr*)

Is true when tree node does not equal *expr*. Otherwise false.

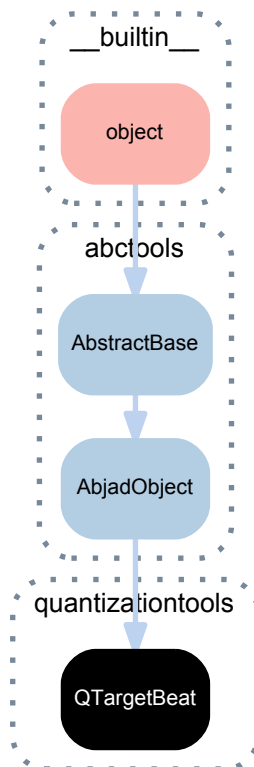
Returns boolean.

(AbjadObject) .**__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

13.2.22 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(
        duration=durationtools.Duration(1, 4),
        units_per_minute=56,
    ),
)
```

Not composer-safe.

Used internally by `Quantizer`.

Bases

- `abctools.AbjadObject`

- `abctools.AbjadObject.AbstractBase`
- `__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=Duration(1, 4), units_per_minute=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).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

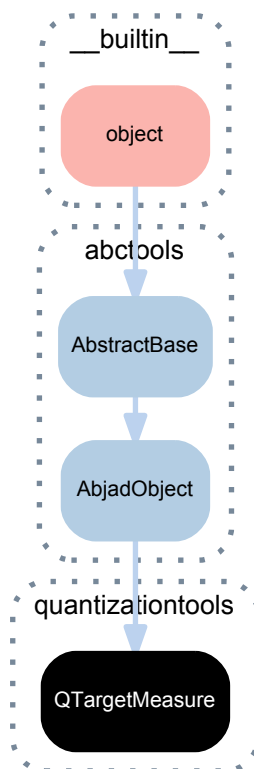
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2.23 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(
    duration=durationtools.Duration(1, 4),
    units_per_minute=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`
- `abctools.AbjadObject.AbstractBase`
- `__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(
        duration=durationtools.Duration(1, 4),
        units_per_minute=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(
        duration=durationtools.Duration(1, 4),
        units_per_minute=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(
        duration=durationtools.Duration(1, 4),
        units_per_minute=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(
        duration=durationtools.Duration(1, 4),
        units_per_minute=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=Duration(1, 4), units_per_minute=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:

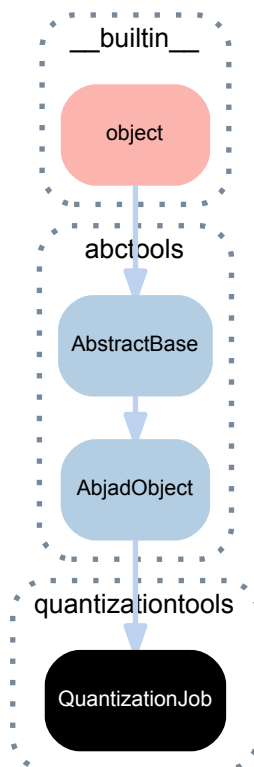
```
>>> q_target_measure.use_full_measure
False
```

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).**__hash__**()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
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.

13.2.24 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`
- `abctools.AbjadObject.AbstractBase`
- `__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(
    quantizationtools.PitchedQEvent(
        offset=durationtools.Offset(250, 1),
        pitches=(
            pitchtools.NamedPitch("c'"),
            pitchtools.NamedPitch("cs'"),
        ),
    ),
    durationtools.Offset(1, 4)
)
quantizationtools.QEventProxy(
```



```

quantizationtools.SilentQEvent(
    offset=durationtools.Offset(500, 1),
),
durationtools.Offset(1, 2)
)
quantizationtools.QEventProxy(
    quantizationtools.PitchedQEvent(
        offset=durationtools.Offset(750, 1),
        pitches=(
            pitchtools.NamedPitch("ef'"),
            pitchtools.NamedPitch("g'"),
        ),
    ),
    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)`

Is 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 Abjad object.

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

Returns string.

`QuantizationJob.__hash__()`

Hashes quantization job.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

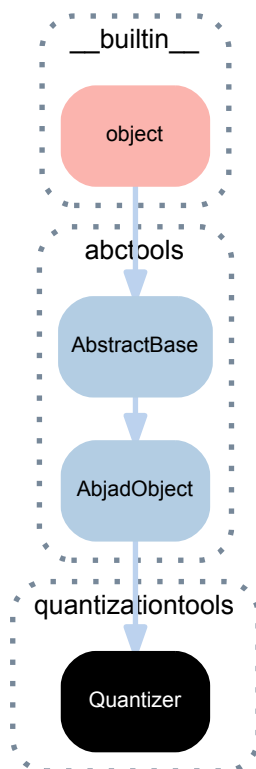
`(AbjadObject).__ne__(expr)`

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

Returns boolean.

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

13.2.25 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(
...     tuple(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
        }
      }
    }
  }
}

```

>>

>>> 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'4
      e'4 ~
      \times 2/3 {
        \tempo 4=30
        e'32
        f'16 ~
      }
      f'16 ~
      \times 2/3 {
        f'32
        fs'16 ~
      }
      fs'16 ~
      \times 2/3 {
        fs'32
        g'16 ~
      }
      g'16 ~
      \times 2/3 {
        g'32
        r16
      }
      r16
    }
  }
>>
```

>>> 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__` ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

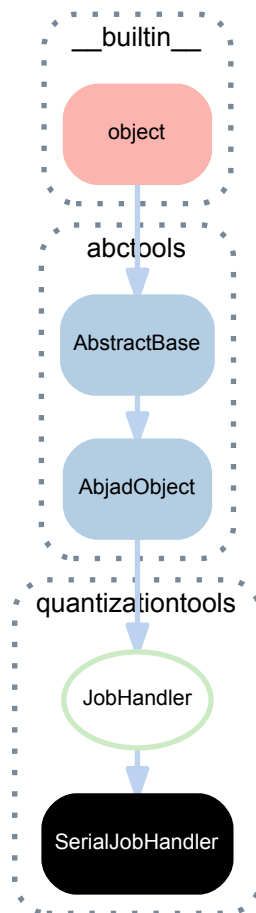
Returns boolean.

`(AbjadObject).__repr__` ()

Gets interpreter representation of Abjad object.

Returns string.

13.2.26 quantizationtools.SerialJobHandler



class `quantizationtools.SerialJobHandler`
Processes `QuantizationJob` instances sequentially.

Bases

- `quantizationtools.JobHandler`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

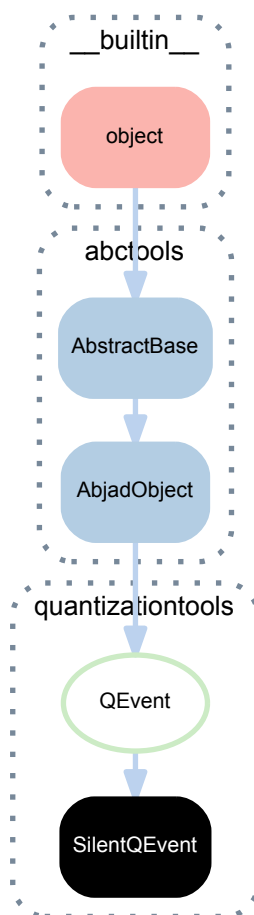
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.2.27 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`
Is true when *expr* is a silen q-event with offset, attachments and index equal to those of this silen q-event. Otherwise false.

Returns boolean.

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

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

Returns string.

`SilentQEvent.__hash__()`
Hashes silen q-event.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(QEvent).__lt__(expr)`
Is true when *expr* is a q-event with offset greater than that of this q-event. Otherwise false.

Returns boolean.

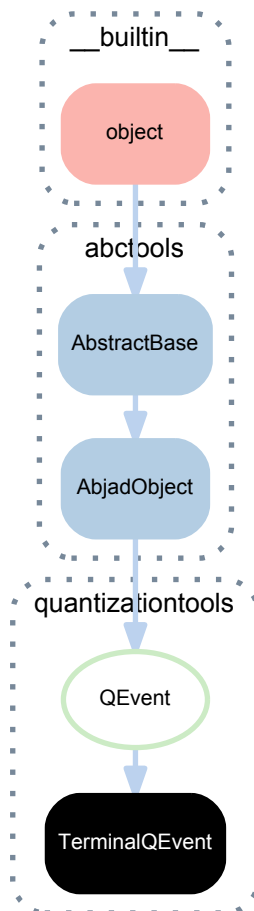
`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

Returns string.

13.2.28 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`TerminalQEvent.__hash__()`

Hashes terminal q-event.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(QEvent).__lt__(expr)`

Is 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.

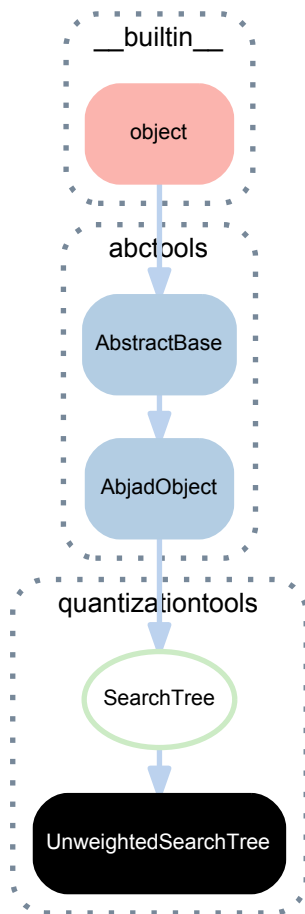
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

13.2.29 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`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

Is 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 Abjad object.

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

Returns string.

(SearchTree).**__hash__**()

Hashes search tree.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

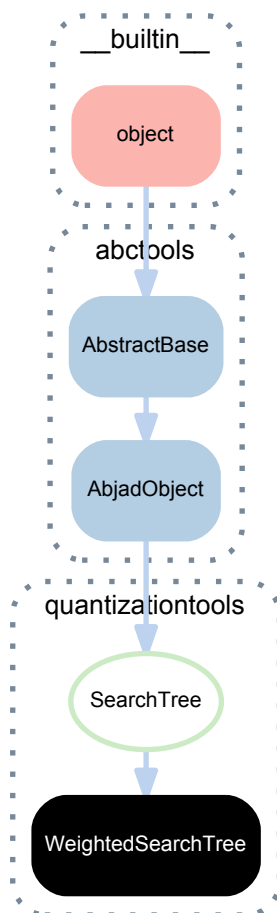
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

13.2.30 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`(SearchTree).__hash__()`

Hashes search tree.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

13.3 Functions

13.3.1 quantizationtools.make_test_time_segments

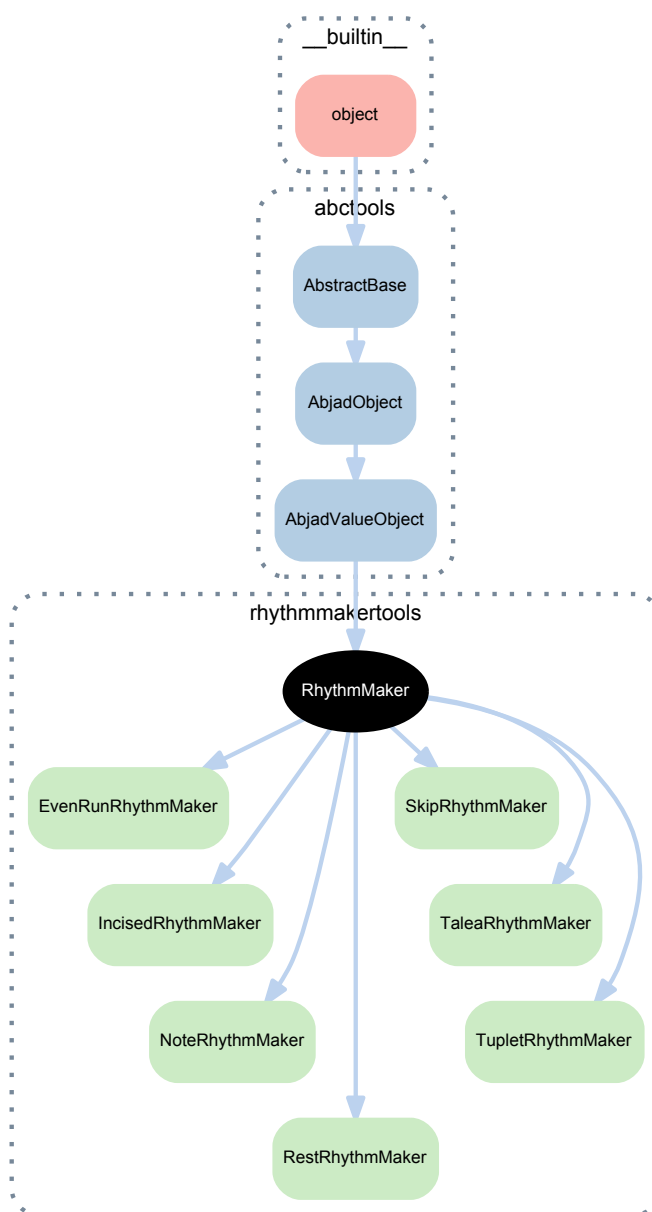
`quantizationtools.make_test_time_segments()`

Make test time segments.

RHYTHMMAKERTOOLS

14.1 Abstract classes

14.1.1 `rhythmmakertools.RhythmMaker`



class `rhythmmakertools.RhythmMaker` (*beam_specifier=None*, *duration_spelling_specifier=None*, *tie_specifier=None*)
 Rhythm-maker abstract base class.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`RhythmMaker.beam_specifier`
 Gets beam specifier of rhythm-maker.
 Returns beam specifier or none.

`RhythmMaker.duration_spelling_specifier`
 Gets duration spelling specifier of rhythm-maker.
 Returns duration spelling specifier or none.

`RhythmMaker.tie_specifier`
 Gets tie specifier of rhythm-maker.
 Return tie specifier or none.

Methods

`RhythmMaker.reverse()`
 Reverses rhythm-maker.
 Concrete rhythm-makers should override this method.
 Returns new rhythm-maker.

Special methods

`RhythmMaker.__call__` (*divisions*, *seeds=None*)
 Calls rhythm-maker.
 Makes music.
 Applies ties specified by tie specifier.
 Checks output type.
 Returns list of selections.

`RhythmMaker.__eq__` (*expr*)
 Is true when *expr* is a rhythm-maker with type and public properties equal to those of this rhythm-maker.
 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.__hash__()`

Hashes rhythm-maker.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`RhythmMaker.__illustrate__(divisions=None)`

Illustrates rhythm-maker.

Defaults *divisions* to 3/8, 4/8, 3/16, 4/16.

Returns LilyPond file.

`(AbjadObject).__ne__(expr)`

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

Returns boolean.

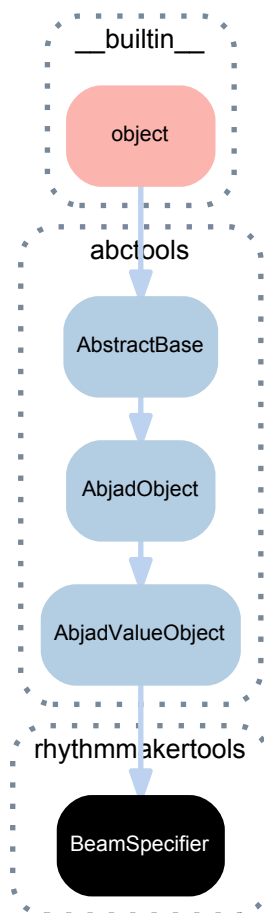
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2 Concrete classes

14.2.1 rhythmmakertools.BeamSpecifier



class `rhythmmakertools.BeamSpecifier` (*beam_each_division=True*,
beam_divisions_together=False)

Beam specifier.

Beam notes in each cell together but do not beam between cells:

```
>>> specifier = rhythmmakertools.BeamSpecifier(
...     beam_each_division=True,
...     beam_divisions_together=False
... )
```

Beam everything:

```
>>> specifier = rhythmmakertools.BeamSpecifier(
...     beam_each_division=True,
...     beam_divisions_together=True
... )
```

Beam nothing:

```
>>> specifier = rhythmmakertools.BeamSpecifier(
...     beam_each_division=True,
...     beam_divisions_together=True
... )
```

Beam specifiers are immutable.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`BeamSpecifier.beam_divisions_together`

Is true when target should beam cells together. Otherwise false.

```
>>> specifier = rhythmmakertools.BeamSpecifier()
>>> specifier.beam_divisions_together
False
```

Defaults to false.

Returns boolean.

`BeamSpecifier.beam_each_division`

Is true when target should beam each cell. Otherwise false.

```
>>> specifier = rhythmmakertools.BeamSpecifier()
>>> specifier.beam_each_division
True
```

Defaults to true.

Returns boolean.

Special methods

`BeamSpecifier.__eq__(arg)`

Is true when *arg* is a beam specifier with *beam_each_division* and *beam_divisions_together* equal to those of this beam specifier. Otherwise false.

```
>>> specifier_1 = rhythmmakertools.BeamSpecifier(
...     beam_each_division=True,
... )
>>> specifier_2 = rhythmmakertools.BeamSpecifier(
...     beam_each_division=False,
... )
```

```
>>> specifier_1 == specifier_1
True
>>> specifier_1 == specifier_2
False
>>> specifier_2 == specifier_1
False
>>> specifier_2 == specifier_2
True
```

Returns boolean.

`BeamSpecifier.__format__` (*format_specification*='')
Formats beam specifier.

```
>>> specifier = rhythmmakertools.BeamSpecifier()
>>> print(format(specifier))
rhythmmakertools.BeamSpecifier(
    beam_each_division=True,
    beam_divisions_together=False,
)
```

Returns string.

`BeamSpecifier.__hash__` ()
Hashes beam specifier.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

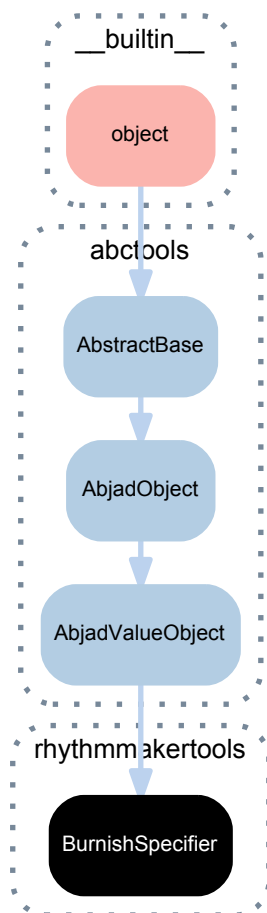
Returns boolean.

`BeamSpecifier.__repr__` ()
Gets interpreter representation of beam specifier.

```
>>> rhythmmakertools.BeamSpecifier()
BeamSpecifier(beam_each_division=True, beam_divisions_together=False)
```

Returns string.

14.2.2 `rhythmmakertools.BurnishSpecifier`



class `rhythmmakertools.BurnishSpecifier` (*burnish_divisions=False, burnish_output=False, lefts=None, middles=None, rights=None, left_lengths=None, right_lengths=None*)

Burnish specifier.

Force first leaf of each division to be a rest:

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1,),
...     left_lengths=(1,),
... )
```

Force the first three leaves of each division to be rests:

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1,),
...     left_lengths=(3,),
... )
```

Force last leaf of each division to be a rest:

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     rights=(-1,),
...     right_lengths=(1,),
... )
```

Force the last three leaves of each division to be rests:

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     rights=(-1,),
...     right_lengths=(3,),
... )
```

Force the first leaf of every even-numbered division to be a rest; force the first leaf of every odd-numbered division to be a note.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 1),
...     left_lengths=(1,),
... )
```

Force the last leaf of every even-numbered division to be a rest; force the last leaf of every odd-numbered division to be a note.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     rights=(-1, 1),
...     right_lengths=(1,),
... )
```

Force the first leaf of every even-numbered division to be a rest; leave the first leaf of every odd-numbered division unchanged.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     left_lengths=(1,),
... )
```

Force the last leaf of every even-numbered division to be a rest; leave the last leaf of every odd-numbered division unchanged.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     rights=(-1, 0),
...     right_lengths=(1,),
... )
```

Burnish specifiers are immutable.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`BurnishSpecifier.burnish_divisions`

Is true when rhythm-maker should burnish every division in output. Otherwise false.

Defaults to false.

Returns boolean.

`BurnishSpecifier.burnish_output`

Is true when rhythm-maker should burnish first and last division in output. Otherwise false.

Defaults to false.

Returns boolean.

`BurnishSpecifier.left_lengths`

Gets left lengths of burnish specifier.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     middles=(0,),
...     rights=(-1, -1, 0),
...     left_lengths=(2,),
```

```
...     right_lengths=(1,),
...     )
```

```
>>> burnish_specifier.left_lengths
(2,)
```

Returns tuple or none.

BurnishSpecifier.lefts

Gets lefts of burnish specifier.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     middles=(0,),
...     rights=(-1, -1, 0),
...     left_lengths=(2,),
...     right_lengths=(1,),
...     )
```

```
>>> burnish_specifier.lefts
(-1, 0)
```

Returns tuple or none.

BurnishSpecifier.middles

Gets middles of burnish specifier.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     middles=(0,),
...     rights=(-1, -1, 0),
...     left_lengths=(2,),
...     right_lengths=(1,),
...     )
```

```
>>> burnish_specifier.middles
(0,)
```

Returns tuple or none.

BurnishSpecifier.right_lengths

Gets right lengths of burnish specifier.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     middles=(0,),
...     rights=(-1, -1, 0),
...     left_lengths=(2,),
...     right_lengths=(1,),
...     )
```

```
>>> burnish_specifier.right_lengths
(1,)
```

Returns tuple or none.

BurnishSpecifier.rights

Gets rights of burnish specifier.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     middles=(0,),
...     rights=(-1, -1, 0),
...     left_lengths=(2,),
...     right_lengths=(1,),
...     )
```

```
>>> burnish_specifier.rights
(-1, -1, 0)
```


Returns tuple or none.

Methods

`BurnishSpecifier.reverse()`

Reverses burnish specification.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     burnish_divisions=True,
...     burnish_output=False,
...     lefts=(-1, 0),
...     middles=(0,),
...     rights=(-1, -1, 0),
...     left_lengths=(2,),
...     right_lengths=(1,),
... )
```

```
>>> print(format(burnish_specifier.reverse()))
rhythmmakertools.BurnishSpecifier(
    burnish_divisions=True,
    burnish_output=False,
    lefts=(0, -1),
    middles=(0,),
    rights=(0, -1, -1),
    left_lengths=(2,),
    right_lengths=(1,),
)
```

Returns new burnish specification.

`BurnishSpecifier.rotate(n=0)`

Rotates burnish specification.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     burnish_divisions=True,
...     burnish_output=False,
...     lefts=(-1, 0),
...     middles=(0,),
...     rights=(-1, -1, 0),
...     left_lengths=(2,),
...     right_lengths=(1, 2, 3),
... )
```

```
>>> print(format(burnish_specifier.rotate(1)))
rhythmmakertools.BurnishSpecifier(
    burnish_divisions=True,
    burnish_output=False,
    lefts=(0, -1),
    middles=(0,),
    rights=(0, -1, -1),
    left_lengths=(2,),
    right_lengths=(3, 1, 2),
)
```

Returns new burnish specification.

Special methods

`BurnishSpecifier.__eq__(expr)`

Is true when *expr* is a burnish specifier with input parameters equal to those of this burnish specifier. Otherwise false.

```
>>> burnish_specifier_1 = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     left_lengths=(1,)
... )
>>> burnish_specifier_2 = rhythmmakertools.BurnishSpecifier(
```

```
...     lefts=(-1, 0),
...     left_lengths=(1,)
... )
>>> burnish_specifier_3 = rhythmmakertools.BurnishSpecifier()
```

```
>>> burnish_specifier_1 == burnish_specifier_1
True
>>> burnish_specifier_1 == burnish_specifier_2
True
>>> burnish_specifier_1 == burnish_specifier_3
False
>>> burnish_specifier_2 == burnish_specifier_1
True
>>> burnish_specifier_2 == burnish_specifier_2
True
>>> burnish_specifier_2 == burnish_specifier_3
False
>>> burnish_specifier_3 == burnish_specifier_1
False
>>> burnish_specifier_3 == burnish_specifier_2
False
>>> burnish_specifier_3 == burnish_specifier_3
True
```

Returns boolean.

`BurnishSpecifier.__format__` (*format_specification*='')
Formats burnish specifier.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     left_lengths=(1,)
... )
```

```
>>> print(format(burnish_specifier))
rhythmmakertools.BurnishSpecifier(
    burnish_divisions=False,
    burnish_output=False,
    lefts=(-1, 0),
    left_lengths=(1,)
)
```

Returns string.

`BurnishSpecifier.__hash__` ()
Hashes burnish specifier.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`BurnishSpecifier.__ne__` (*expr*)
Is true when *expr* does not equal burnish specifier.

```
>>> burnish_specifier_1 = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     left_lengths=(1,)
... )
>>> burnish_specifier_2 = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     left_lengths=(1,)
... )
>>> burnish_specifier_3 = rhythmmakertools.BurnishSpecifier()
```

```
>>> burnish_specifier_1 != burnish_specifier_1
False
>>> burnish_specifier_1 != burnish_specifier_2
False
>>> burnish_specifier_1 != burnish_specifier_3
True
>>> burnish_specifier_2 != burnish_specifier_1
```

```
False
>>> burnish_specifier_2 != burnish_specifier_2
False
>>> burnish_specifier_2 != burnish_specifier_3
True
>>> burnish_specifier_3 != burnish_specifier_1
True
>>> burnish_specifier_3 != burnish_specifier_2
True
>>> burnish_specifier_3 != burnish_specifier_3
False
```

Returns boolean.

`BurnishSpecifier.__repr__()`

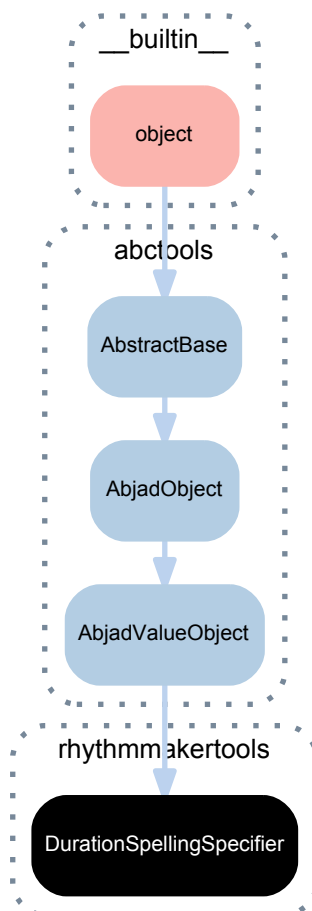
Gets interpreter representation of burnish specifier.

```
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     lefts=(-1, 0),
...     left_lengths=(1,),
... )
```

```
>>> burnish_specifier
BurnishSpecifier(burnish_divisions=False, burnish_output=False, lefts=(-1, 0), left_lengths=(1,))
```

Returns string.

14.2.3 `rhythmmakertools.DurationSpellingSpecifier`



class `rhythmmakertools.DurationSpellingSpecifier` (*decrease_durations_monotonically=True*,
forbidden_written_duration=None)
 Duration spelling specifier.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`DurationSpellingSpecifier.decrease_durations_monotonically`

Is true when all durations should be spelled as a tied series of monotonically decreasing values. Otherwise false.

```
>>> specifier = rhythmmakertools.DurationSpellingSpecifier()
>>> specifier.decrease_durations_monotonically
True
```

Defaults to true.

Returns boolean.

`DurationSpellingSpecifier.forbidden_written_duration`

Gets forbidden written duration.

```
>>> specifier = rhythmmakertools.DurationSpellingSpecifier()
>>> specifier.forbidden_written_duration is None
True
```

Defaults to none.

Returns duration or none.

Methods

`DurationSpellingSpecifier.reverse()`

Reverses duration spelling specifier.

```
>>> specifier = rhythmmakertools.DurationSpellingSpecifier()
>>> print(format(specifier))
rhythmmakertools.DurationSpellingSpecifier(
    decrease_durations_monotonically=True,
)
```

```
>>> reversed_specifier = specifier.reverse()
>>> print(format(reversed_specifier))
rhythmmakertools.DurationSpellingSpecifier(
    decrease_durations_monotonically=False,
)
```

Negates *decrease_monotonically*.

Returns new duration spelling specifier.

Special methods

`DurationSpellingSpecifier.__eq__(arg)`

Is true when *arg* is a duration spelling specifier with values of *decrease_durations_monotonically* and *forbidden_written_duration* equal to those of this duration spelling specifier. Otherwise false.

```
>>> specifier_1 = rhythmmakertools.DurationSpellingSpecifier(
...     decrease_durations_monotonically=True,
... )
>>> specifier_2 = rhythmmakertools.DurationSpellingSpecifier(
...     decrease_durations_monotonically=False,
... )
```

```
>>> specifier_1 == specifier_1
True
>>> specifier_1 == specifier_2
False
>>> specifier_2 == specifier_1
False
>>> specifier_2 == specifier_2
True
```

Returns boolean.

`DurationSpellingSpecifier.__format__` (*format_specification*='')
Formats duration spelling specifier.

```
>>> specifier = rhythmmakertools.DurationSpellingSpecifier()
>>> print(format(specifier))
rhythmmakertools.DurationSpellingSpecifier(
    decrease_durations_monotonically=True,
)
```

Returns string.

`DurationSpellingSpecifier.__hash__` ()
Hashes duration spelling specifier.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

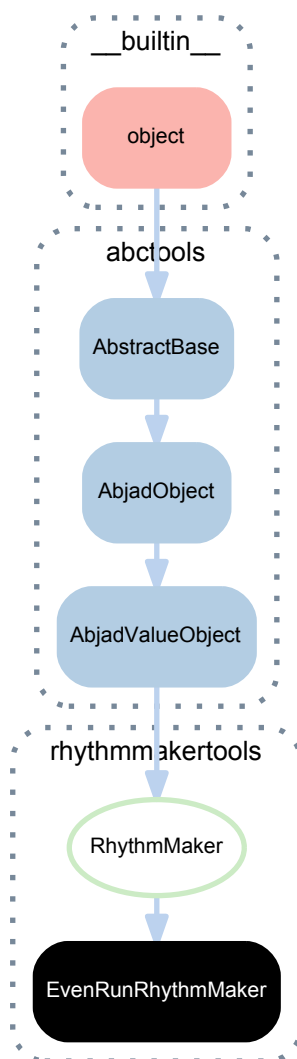
`(AbjadObject).__ne__` (*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`DurationSpellingSpecifier.__repr__` ()
Gets interpreter representation of duration spelling specifier.

```
>>> rhythmmakertools.DurationSpellingSpecifier()
DurationSpellingSpecifier(decrease_durations_monotonically=True)
```

Returns string.

14.2.4 `rhythmmakertools.EvenRunRhythmMaker`

class `rhythmmakertools.EvenRunRhythmMaker` (*exponent=None*, *beam_specifier=None*,
duration_spelling_specifier=None,
tie_specifier=None)

Even run rhythm-maker.

Makes 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)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
  
```



Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

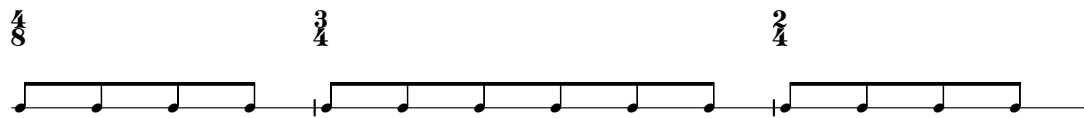
`(RhythmMaker).beam_specifier`
Gets beam specifier of rhythm-maker.

Returns beam specifier or none.

`EvenRunRhythmMaker.duration_spelling_specifier`
Gets duration spelling specifier of even-run rhythm-maker.

```
>>> specifier = rhythmmakertools.DurationSpellingSpecifier(
...     forbidden_written_duration=Duration(1, 4),
... )
>>> maker = rhythmmakertools.EvenRunRhythmMaker(
...     duration_spelling_specifier=specifier,
... )
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Returns duration spelling specifier or none.

`EvenRunRhythmMaker.exponent`
Gets exponent of even-run rhythm-maker.

Makes 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(
...     exponent=1,
... )
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Defaults to none and interprets none equal to 0.

Returns nonnegative integer or none.

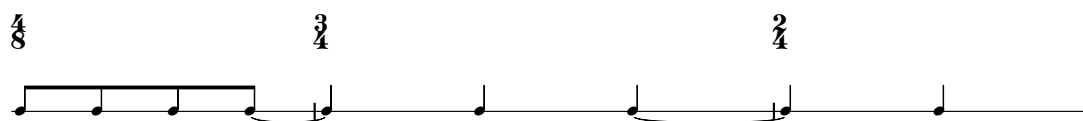
EvenRunRhythmMaker.tie_specifier

Gets tie specifier of rhythm-maker.

Ties across divisions:

```
>>> tie_specifier = rhythmmakertools.TieSpecifier(
...     tie_across_divisions=True,
... )
>>> maker = rhythmmakertools.EvenRunRhythmMaker(
...     tie_specifier=tie_specifier,
... )

>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Returns boolean.

Methods

EvenRunRhythmMaker.reverse()

Reverses even-run rhythm-maker.

```
>>> maker = rhythmmakertools.EvenRunRhythmMaker()
>>> reversed_maker = maker.reverse()

>>> print(format(reversed_maker))
rhythmmakertools.EvenRunRhythmMaker(
    duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(
        decrease_durations_monotonically=False,
    ),
)

>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Returns new even-run rhythm-maker.

Special methods

EvenRunRhythmMaker.__call__(divisions, seeds=None)

Calls even-run rhythm-maker on *divisions*.

```
>>> maker = rhythmmakertools.EvenRunRhythmMaker()
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> result = maker(divisions)
>>> for selection in result:
...     selection
```



```
Selection(Container("c'8 c'8 c'8 c'8"),)
Selection(Container("c'4 c'4 c'4"),)
Selection(Container("c'4 c'4"),)
```

Returns a list of selections. Each selection holds a single container filled with notes.

(RhythmMaker).**__eq__**(*expr*)

Is true when *expr* is a rhythm-maker with type and public properties equal to those of this rhythm-maker. Otherwise false.

Returns boolean.

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

Formats even run rhythm-maker.

```
>>> print(format(maker))
rhythmmakertools.EvenRunRhythmMaker()
```

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

Returns string.

(RhythmMaker).**__hash__**()

Hashes rhythm-maker.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(RhythmMaker).**__illustrate__**(*divisions=None*)

Illustrates rhythm-maker.

Defaults *divisions* to 3/8, 4/8, 3/16, 4/16.

Returns LilyPond file.

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

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

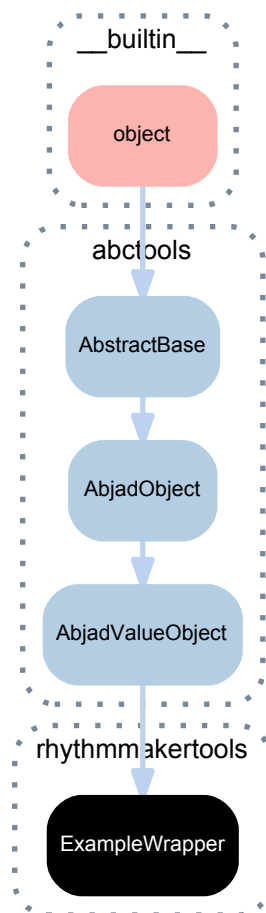
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.5 rhythmtools.ExampleWrapper



class `rhythmtools.ExampleWrapper` (*arguments=None, division_lists=None*)
 Example wrapper.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`ExampleWrapper.arguments`
 Gets arguments of example wrapper.
 Returns dictionary.

`ExampleWrapper.division_lists`
 Gets division lists of example wrapper.
 Returns tuple.

Special methods

`(AbjadValueObject).__eq__(expr)`

Is true when all initialization values of Abjad value object equal the initialization values of *expr*.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`(AbjadValueObject).__hash__()`

Hashes Abjad value object.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

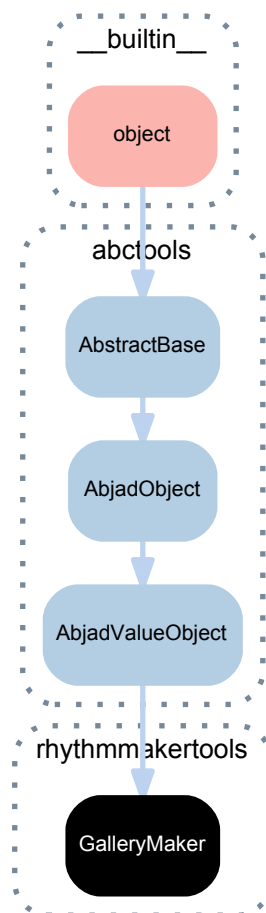
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.6 `rhythmmakertools.GalleryMaker`



class `rhythmmakertools.GalleryMaker`
 Gallery maker.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Special methods

`GalleryMaker.__call__(configurations_by_class)`

Calls gallery-maker.

Returns LilyPond file.

`(AbjadValueObject).__eq__(expr)`

Is true when all initialization values of Abjad value object equal the initialization values of *expr*.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`(AbjadValueObject).__hash__()`

Hashes Abjad value object.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

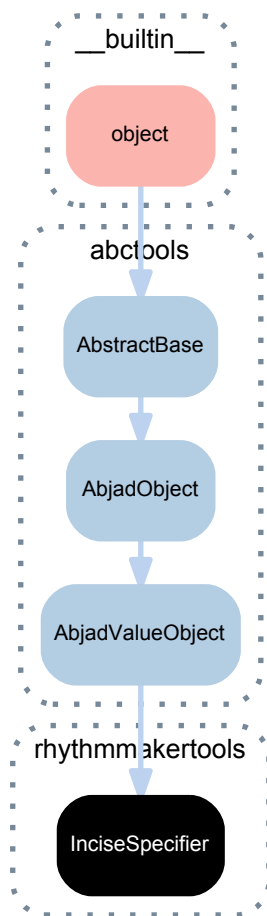
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.7 rhythmmakertools.InciseSpecifier



class `rhythmmakertools.InciseSpecifier` (*incise_divisions=False*, *incise_output=False*,
prefix_talea=(-1,), *prefix_lengths=(0, 1)*,
suffix_talea=(-11,), *suffix_lengths=(1,)*,
talea_denominator=32, *body_ratio=None*,
fill_with_notes=True)

Incision specifier.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`InciseSpecifier.body_ratio`
 Gets body ratio of incise specifier.

Sets *body_ratio* to divide middle part proportionally:

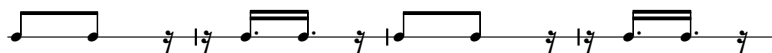
```

>>> incise_specifier = rhythmmakertools.InciseSpecifier(
...     incise_divisions=True,
...     prefix_talea=(-1,),
...     prefix_lengths=(0, 1),
...     suffix_talea=(-1,),
  
```

```
...     suffix_lengths=(1, ),
...     talea_denominator=16,
...     body_ratio=(1, 1),
...     )
>>> maker = rhythmmakertools.IncisedRhythmMaker(
...     incise_specifier=incise_specifier,
...     )

>>> divisions = 4 * [(5, 16)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```

5
16



Defaults to none.

Returns ratio or none.

InciseSpecifier.fill_with_notes

Is true when rhythm-maker should fill divisions with notes. Otherwise false.

Defaults to true.

Returns boolean.

InciseSpecifier.incise_divisions

Is true when rhythm-maker should incise every division. Otherwise false.

Defaults to false.

Returns boolean.

InciseSpecifier.incise_output

Is true when rhythm-maker should incise first and last divisions. Otherwise false.

Defaults to false.

Returns boolean.

InciseSpecifier.prefix_lengths

Gets prefix lengths of incision specifier.

Returns tuple or none.

InciseSpecifier.prefix_talea

Gets prefix talea of incision specifier.

Returns tuple or none.

InciseSpecifier.suffix_lengths

Gets suffix lengths of incision specifier.

Returns tuple or none.

InciseSpecifier.suffix_talea

Gets suffix talea of incision specifier.

Returns tuple or none.

InciseSpecifier.talea_denominator

Gets talea denominator of incision specifier.

Returns positive integer-equivalent number.

Methods

`InciseSpecifier.reverse()`
Reverses incision specifier.

Returns new incision specifier.

`InciseSpecifier.rotate(n=0)`
Rotates incision specifier.

```
>>> incise_specifier = rhythmmakertools.InciseSpecifier(
...     incise_divisions=True,
...     prefix_talea=(-1,),
...     prefix_lengths=(0, 2, 1),
...     suffix_talea=(-1, 1),
...     suffix_lengths=(1, 0, 0),
...     talea_denominator=16,
...     body_ratio=(1, 1),
... )
```

```
>>> print(format(incise_specifier.rotate(1)))
rhythmmakertools.InciseSpecifier(
    incise_divisions=True,
    incise_output=False,
    prefix_talea=(-1,),
    prefix_lengths=(1, 0, 2),
    suffix_talea=(1, -1),
    suffix_lengths=(0, 1, 0),
    talea_denominator=16,
    body_ratio=mathtools.Ratio(1, 1),
    fill_with_notes=True,
)
```

Returns new incision specifier.

Special methods

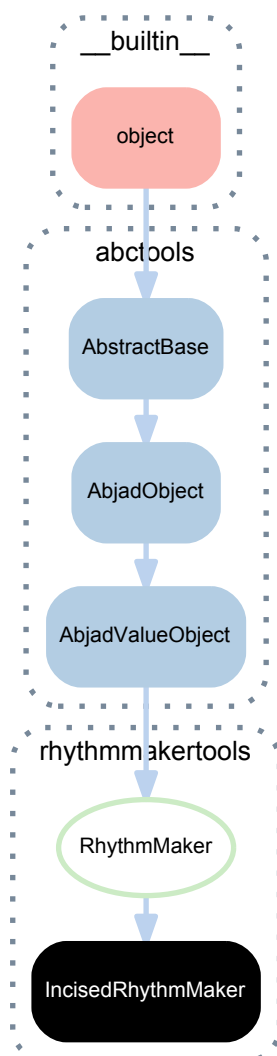
`(AbjadValueObject).__eq__(expr)`
Is true when all initialization values of Abjad value object equal the initialization values of *expr*.
Returns boolean.

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

`(AbjadValueObject).__hash__()`
Hashes Abjad value object.
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.

14.2.8 `rhythmmakertools.IncisedRhythmMaker`

```

class rhythmmakertools.IncisedRhythmMaker (incise_specifier=None,
                                           split_divisions_by_counts=None,
                                           extra_counts_per_division=None,
                                           beam_specifier=None,           dura-
                                           tion_spelling_specifier=None,
                                           tie_specifier=None, helper_functions=None)

```

Incised rhythm-maker.

```

>>> incise_specifier = rhythmmakertools.InciseSpecifier(
...     incise_divisions=True,
...     prefix_talea=(-1,),
...     prefix_lengths=(0, 1),
...     suffix_talea=(-1,),
...     suffix_lengths=(1,),
...     talea_denominator=16,
... )
>>> maker = rhythmmakertools.IncisedRhythmMaker(
...     incise_specifier=incise_specifier,
... )

```

```

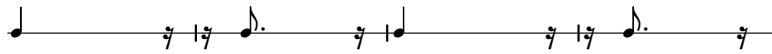
>>> divisions = 4 * [(5, 16)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )

```



```
>>> show(lilypond_file)
```

5
16



Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(RhythmMaker).beam_specifier`
Gets beam specifier of rhythm-maker.

Returns beam specifier or none.

`(RhythmMaker).duration_spelling_specifier`
Gets duration spelling specifier of rhythm-maker.

Returns duration spelling specifier or none.

`IncisedRhythmMaker.extra_counts_per_division`
Gets prolation addenda of incised rhythm-maker.

Returns tuple or none.

`IncisedRhythmMaker.helper_functions`
Gets helper functions of incised rhythm-maker.

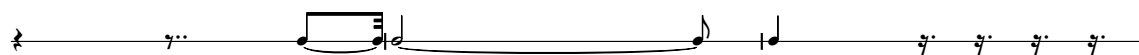
Returns dictionary or none.

`IncisedRhythmMaker.incise_specifier`
Gets incise specifier or incised rhythm-maker.

Output-incised notes:

```
>>> incise_specifier = rhythmmakertools.InciseSpecifier(
...     incise_output=True,
...     prefix_talea=(-8, -7),
...     prefix_lengths=(2,),
...     suffix_talea=(-3,),
...     suffix_lengths=(4,),
...     talea_denominator=32,
... )
>>> maker = rhythmmakertools.IncisedRhythmMaker(
...     incise_specifier=incise_specifier,
... )
```

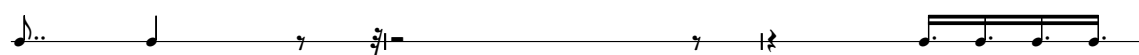
```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```

5
8

Output-incised rests:

```
>>> incise_specifier = rhythmmakertools.InciseSpecifier(
...     incise_output=True,
...     prefix_talea=(7, 8),
...     prefix_lengths=(2,),
...     suffix_talea=(3,),
...     suffix_lengths=(4,),
...     talea_denominator=32,
...     fill_with_notes=False,
... )
>>> maker = rhythmmakertools.IncisedRhythmMaker(
...     incise_specifier=incise_specifier,
... )
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```

5
8

Returns incise specifier or none.

IncisedRhythmMaker.split_divisions_by_counts

Gets secondary divisions of incised rhythm-maker.

Returns tuple or none.

(RhythmMaker).tie_specifier

Gets tie specifier of rhythm-maker.

Return tie specifier or none.

Methods

IncisedRhythmMaker.reverse()

Reverses incised rhythm-maker.

Returns newly constructed rhythm-maker.

IncisedRhythmMaker.rotate(n=0)

Rotates incised rhythm-maker.

Returns newly constructed rhythm-maker.

Special methods

IncisedRhythmMaker.__call__(divisions, seeds=None)

Calls incised rhythm-maker on *divisions*.

Returns list of selections.

(RhythmMaker).__eq__(expr)

Is true when *expr* is a rhythm-maker with type and public properties equal to those of this rhythm-maker. 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).**__hash__**()

Hashes rhythm-maker.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(RhythmMaker).**__illustrate__**(*divisions*=None)

Illustrates rhythm-maker.

Defaults *divisions* to 3/8, 4/8, 3/16, 4/16.

Returns LilyPond file.

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

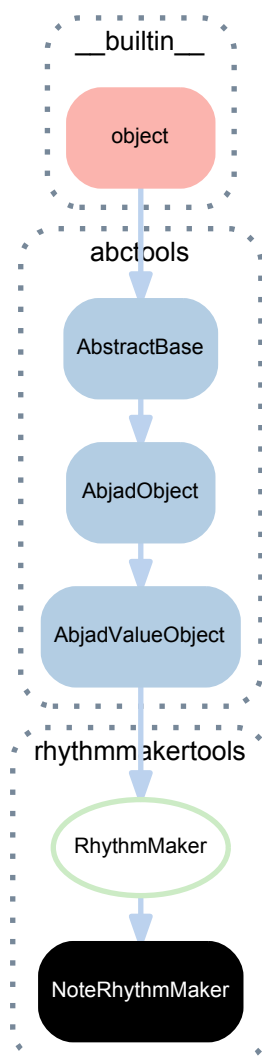
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.9 `rhythmmakertools.NoteRhythmMaker`

class `rhythmmakertools.NoteRhythmMaker` (*beam_specifier=None*, *duration_spelling_specifier=None*, *tie_specifier=None*)

Note rhythm-maker.

Makes notes equal to the duration of input divisions. Adds ties where necessary:

```

>>> maker = rhythmmakertools.NoteRhythmMaker()

>>> divisions = [(5, 8), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
  
```



Usage follows the two-step configure-once / call-repeatedly pattern shown here.

Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(RhythmMaker).beam_specifier`
Gets beam specifier of rhythm-maker.

Returns beam specifier or none.

`NoteRhythmMaker.duration_spelling_specifier`
Gets duration spelling specifier of note rhythm-maker.

Forbids notes with written duration greater than or equal to 1/2 of a whole note:

```
>>> duration_spelling_specifier = \
...     rhythmmakertools.DurationSpellingSpecifier(
...         forbidden_written_duration=Duration(1, 2),
...     )
>>> maker = rhythmmakertools.NoteRhythmMaker(
...     duration_spelling_specifier=duration_spelling_specifier,
... )
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Returns duration spelling specifier or none.

`(RhythmMaker).tie_specifier`
Gets tie specifier of rhythm-maker.

Return tie specifier or none.

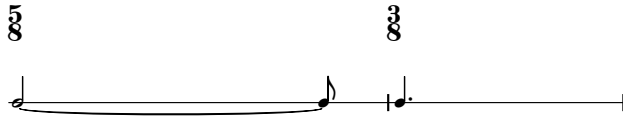
Methods

`NoteRhythmMaker.reverse()`
Reverses note rhythm-maker.

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
>>> reversed_maker = maker.reverse()
```

```
>>> print(format(reversed_maker))
rhythmmakertools.NoteRhythmMaker(
    duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(
        decrease_durations_monotonically=False,
    ),
)
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```



Defined equal to copy of note rhythm-maker with *duration_spelling_specifier* reversed.

Returns new note rhythm-maker.

Special methods

`NoteRhythmMaker.__call__(divisions, seeds=None)`

Calls note rhythm-maker on *divisions*.

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
>>> divisions = [(5, 8), (3, 8)]
>>> result = maker(divisions)
>>> for x in result:
...     x
Selection(Note("c'2"), Note("c'8"))
Selection(Note("c'4."),)
```

Returns list of selections. Each selection holds one or more notes.

`NoteRhythmMaker.__eq__(arg)`

True when *arg* is a note rhythm-maker with values of *beam_specifier*, *duration_spelling_specifier* and *tie_specifier* equal to those of this note rhythm-maker. Otherwise false.

```
>>> maker_1 = rhythmmakertools.NoteRhythmMaker()
>>> tie_specifier = rhythmmakertools.TieSpecifier(
...     tie_across_divisions=True,
...     )
>>> maker_2 = rhythmmakertools.NoteRhythmMaker(
...     tie_specifier=tie_specifier,
...     )
```

```
>>> maker_1 == maker_1
True
>>> maker_1 == maker_2
False
>>> maker_2 == maker_1
False
>>> maker_2 == maker_2
True
```

Returns boolean.

`NoteRhythmMaker.__format__(format_specification='')`

Formats note rhythm-maker.

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

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
>>> print(format(maker))
rhythmmakertools.NoteRhythmMaker()
```

Returns string.

`NoteRhythmMaker.__hash__()`

Hashes note rhythm-maker.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(RhythmMaker).__illustrate__ (divisions=None)`

Illustrates rhythm-maker.

Defaults *divisions* to 3/8, 4/8, 3/16, 4/16.

Returns LilyPond file.

`(AbjadObject).__ne__ (expr)`

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

Returns boolean.

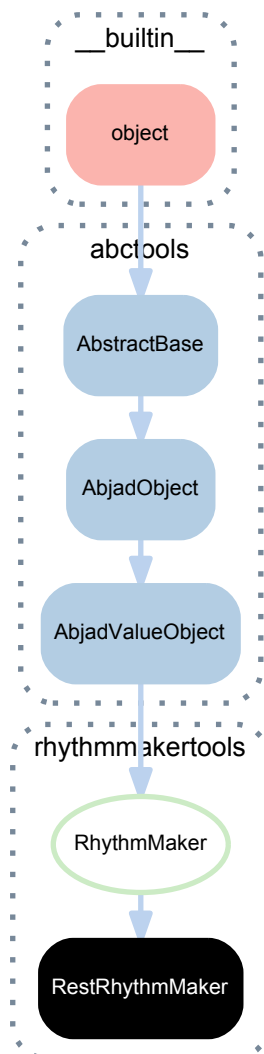
`NoteRhythmMaker.__repr__ ()`

Gets interpreter representation of note rhythm-maker.

```
>>> rhythmmakertools.NoteRhythmMaker ()
NoteRhythmMaker ()
```

Returns string.

14.2.10 rhythmmakertools.RestRhythmMaker



class `rhythmmakertools.RestRhythmMaker` (*duration_spelling_specifier=None*)
 Rest rhythm-maker.

Makes rests equal to the duration of input divisions.

```
>>> maker = rhythmmakertools.RestRhythmMaker()

>>> divisions = [(5, 16), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```

The image shows the output of the Python code. It displays two time signatures, $\frac{5}{16}$ and $\frac{3}{8}$, followed by a musical staff. The staff contains a single eighth rest, represented by a vertical line with a diagonal slash and a flag.

Usage follows the two-step configure-once / call-repeatedly pattern shown here.

Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(RhythmMaker).**beam_specifier**
Gets beam specifier of rhythm-maker.

Returns beam specifier or none.

RestRhythmMaker.**duration_spelling_specifier**
Gets duration spelling specifier of rest rhythm-maker.

Forbids rests with written duration greater than or equal to 1/4 of a whole note:

```
>>> duration_spelling_specifier = \
...     rhythmmakertools.DurationSpellingSpecifier(
...     forbidden_written_duration=Duration(1, 4),
...     )
>>> maker = rhythmmakertools.RestRhythmMaker(
...     duration_spelling_specifier=duration_spelling_specifier,
...     )
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```

The image shows the output of the Python code. It displays two time signatures, $\frac{5}{16}$ and $\frac{3}{8}$, followed by a musical staff. The staff contains a beamed eighth rest, represented by a vertical line with a diagonal slash and a flag, and a beam connecting it to the next note.

Returns duration spelling specifier or none.

`(RhythmMaker).tie_specifier`
 Gets tie specifier of rhythm-maker.
 Return tie specifier or none.

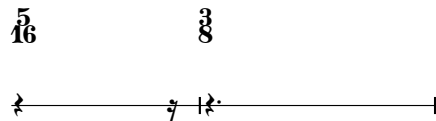
Methods

`RestRhythmMaker.reverse()`
 Reverses rest rhythm-maker.

```
>>> maker = rhythmmakertools.RestRhythmMaker()
>>> reversed_maker = maker.reverse()
```

```
>>> print(format(reversed_maker))
rhythmmakertools.RestRhythmMaker(
    duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(
        decrease_durations_monotonically=False,
    ),
)
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```



Returns new rest rhythm-maker.

Special methods

`RestRhythmMaker.__call__(divisions, seeds=None)`
 Calls rest rhythm-maker on *divisions*.
 Returns list of selections.

`(RhythmMaker).__eq__(expr)`
 Is true when *expr* is a rhythm-maker with type and public properties equal to those of this rhythm-maker.
 Otherwise false.
 Returns boolean.

`RestRhythmMaker.__format__(format_specification='')`
 Formats rest rhythm-maker.
 Set *format_specification* to `'` or `'storage'`.

```
>>> print(format(maker))
rhythmmakertools.RestRhythmMaker()
```

Returns string.

`(RhythmMaker).__hash__()`
 Hashes rhythm-maker.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 Returns integer.

(RhythmMaker).**__illustrate__**(*divisions=None*)

Illustrates rhythm-maker.

Defaults *divisions* to 3/8, 4/8, 3/16, 4/16.

Returns LilyPond file.

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

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

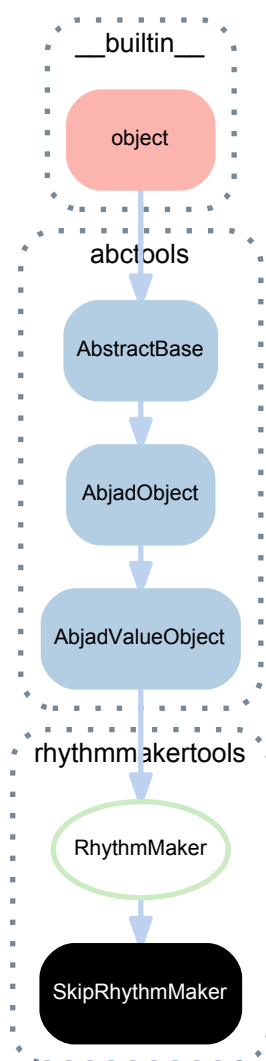
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.11 rhythm makertools.SkipRhythmMaker



class rhythm makertools.**SkipRhythmMaker**

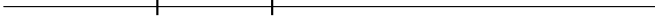
Skip rhythm-maker.

Makes skips equal to the duration of input divisions.

```
>>> maker = rhythm makertools.SkipRhythmMaker()
```

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```

$\frac{1}{4}$ $\frac{3}{16}$ $\frac{5}{8}$



Usage follows the two-step configure-once / call-repeatedly pattern shown here.

Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(RhythmMaker).beam_specifier`
Gets beam specifier of rhythm-maker.

Returns beam specifier or none.

`(RhythmMaker).duration_spelling_specifier`
Gets duration spelling specifier of rhythm-maker.

Returns duration spelling specifier or none.

`(RhythmMaker).tie_specifier`
Gets tie specifier of rhythm-maker.

Return tie specifier or none.

Methods


`SkipRhythmMaker.reverse()`
Reverses skip rhythm-maker.

```
>>> reversed_maker = maker.reverse()
```

```
>>> print(format(reversed_maker))
rhythmmakertools.SkipRhythmMaker()
```

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```

$\frac{1}{4}$ $\frac{3}{16}$ $\frac{5}{8}$



Defined equal to copy of rhythm-maker.

Returns new skip rhythm-maker.

Special methods

`SkipRhythmMaker.__call__(divisions, seeds=None)`

Calls skip rhythm-maker on *divisions*.

Returns list of selections.

`(RhythmMaker).__eq__(expr)`

Is true when *expr* is a rhythm-maker with type and public properties equal to those of this rhythm-maker. 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.

`(RhythmMaker).__hash__()`

Hashes rhythm-maker.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(RhythmMaker).__illustrate__(divisions=None)`

Illustrates rhythm-maker.

Defaults *divisions* to 3/8, 4/8, 3/16, 4/16.

Returns LilyPond file.

`(AbjadObject).__ne__(expr)`

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

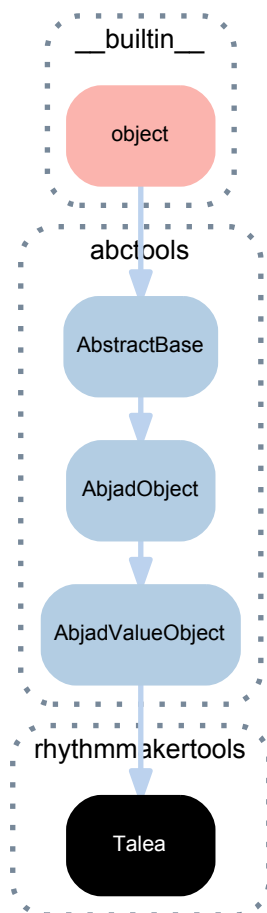
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.12 `rhythmmakertools.Talea`



class `rhythmmakertools.Talea` (*counts*=(1,), *denominator*=16)
 Talea.

```

>>> talea = rhythmmakertools.Talea(
...     counts=(2, 1, 3, 2, 4, 1, 1),
...     denominator=16,
... )
  
```

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`Talea.counts`
 Gets counts of talea.
 Returns tuple.

`Talea.denominator`
 Gets denominator of talea.
 Returns nonnegative integer power of two.

Methods

`Talea.reverse()`

Reverses talea.

```
>>> reversed_talea = talea.reverse()
>>> print(format(reversed_talea))
rhythmmakertools.Talea(
  counts=(1, 1, 4, 2, 3, 1, 2),
  denominator=16,
)
```

Returns new talea.

`Talea.rotate(n=0)`

Rotates talea by *n*.

```
>>> rotated_talea = talea.rotate(1)
>>> print(format(rotated_talea))
rhythmmakertools.Talea(
  counts=(1, 2, 1, 3, 2, 4, 1),
  denominator=16,
)
```

Returns new talea.

Special methods

`Talea.__eq__(expr)`

Is true when *expr* is a talea with *counts* and *denominator* equal to those of this talea. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`Talea.__hash__()`

Hashes talea.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

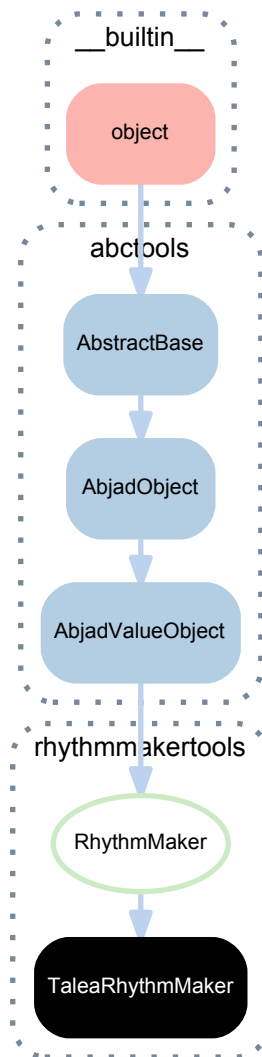
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

14.2.13 rhythmtools.TaleaRhythmMaker



```

class rhythmtools.TaleaRhythmMaker (talea=None,    split_divisions_by_counts=None,
                                     extra_counts_per_division=None,
                                     beam_specifier=None,    burnish_specifier=None,
                                     duration_spelling_specifier=None,
                                     tie_specifier=None, helper_functions=None)

```

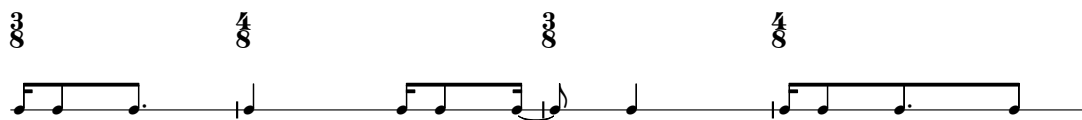
Talea rhythm-maker.

```

>>> talea = rhythmtools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> maker = rhythmtools.TaleaRhythmMaker(
...     talea=talea,
... )

>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmtools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)

```



Follows the two-step configure-once / call-repeatedly pattern shown here.

Object model of a partially evaluated function that accepts a (possibly empty) list of divisions as input and returns a list of selections as output (structured one selection per input division).

Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`TaleaRhythmMaker.beam_specifier`

Gets beam specifier of talea rhythm-maker.

Three beam specifier configurations are available.

This rhythm-maker beams each division:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1,),
...     denominator=16,
... )
>>> beam_specifier = rhythmmakertools.BeamSpecifier(
...     beam_each_division=True,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     beam_specifier=beam_specifier,
... )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



The behavior shown here is the talea rhythm-maker's default beaming.

This rhythm-maker beams divisions together:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1,),
...     denominator=16,
... )
>>> beam_specifier = rhythmmakertools.BeamSpecifier(
...     beam_divisions_together=True,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
```



```
...     beam_specifier=beam_specifier,
...     )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```



This rhythm-maker makes no beams:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1,),
...     denominator=16,
...     )
>>> beam_specifier = rhythmmakertools.BeamSpecifier(
...     beam_each_division=False,
...     beam_divisions_together=False,
...     )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     beam_specifier=beam_specifier,
...     )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```



Returns beam specifier or none.

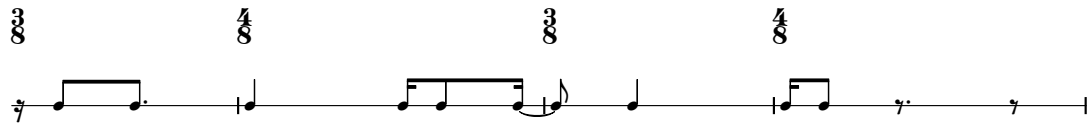
`TaleaRhythmMaker.burnish_specifier`
Gets burnish specifier of talea rhythm-maker.

‘Output burnishing’ means forcibly to cast the first leaf (or leaves) of output; or forcibly to cast the last leaf (or leaves) of output; or to cast both the first and last leaves of output at the same time.

This example makes a talea rhythm with the first leaf of output forcibly cast to a rest and also with the last two leaves of output forcibly cast to rests:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
...     )
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     burnish_output=True,
...     lefts=(-1,),
...     middles=(0,),
...     rights=(-1,),
...     left_lengths=(1,),
...     right_lengths=(2,),
...     )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     burnish_specifier=burnish_specifier,
...     )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



‘Division burnishing’ means forcibly to cast the first leaf (or leaves) of every division; or forcibly to cast the last leaf (or leaves) of every division; or forcibly to cast both the first and last leaves of every division at the same time.

This example makes a talea rhythm with the first leaf of every division forcibly cast to a rest:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> burnish_specifier = rhythmmakertools.BurnishSpecifier(
...     burnish_divisions=True,
...     lefts=(-1,),
...     middles=(0,),
...     rights=(0,),
...     left_lengths=(1,),
...     right_lengths=(0,),
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     burnish_specifier=burnish_specifier,
... )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Returns burnish specifier or none.

TaleaRhythmMaker.duration_spelling_specifier

Gets duration spelling specifier of talea rhythm-maker.

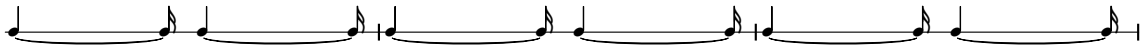
Several beam spelling specifier configurations are available.

This rhythm-maker spells nonassignable durations like 5/16 with monotonically decreasing durations:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(5,),
...     denominator=16,
... )
>>> duration_spelling_specifier = \
...     rhythmmakertools.DurationSpellingSpecifier(
...         decrease_durations_monotonically=True,
...     )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     beam_specifier=beam_specifier,
... )
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```

$\frac{5}{8}$



The behavior shown here is a default duration-spelling behavior.

This rhythm-maker spells nonassignable durations like 5/16 with monotonically increasing durations:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(5,),
...     denominator=16,
...     )
>>> duration_spelling_specifier = \
...     rhythmmakertools.DurationSpellingSpecifier(
...         decrease_durations_monotonically=False,
...         )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     duration_spelling_specifier=duration_spelling_specifier,
...     )
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```

$\frac{5}{8}$



This rhythm-maker has no forbidden durations:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 1, 1, 1, 4, 4),
...     denominator=16,
...     )
>>> duration_spelling_specifier = \
...     rhythmmakertools.DurationSpellingSpecifier(
...         forbidden_written_duration=None,
...         )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     duration_spelling_specifier=duration_spelling_specifier,
...     )
```

```
>>> divisions = [(3, 4), (3, 4)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
...     )
>>> show(lilypond_file)
```

$\frac{3}{4}$



The behavior shown here is a default duration-spelling behavior.

This rhythm-maker forbids durations equal to 1/4 or greater:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 1, 1, 1, 4, 4),
...     denominator=16,
... )
>>> duration_spelling_specifier = \
...     rhythmmakertools.DurationSpellingSpecifier(
...         forbidden_written_duration=Duration(1, 4),
...     )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     duration_spelling_specifier=duration_spelling_specifier,
... )
```

```
>>> divisions = [(3, 4), (3, 4)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Forbidden durations are rewritten with smaller durations tied together.

Returns duration spelling specifier or none.

TaleaRhythmMaker.extra_counts_per_division

Gets extra counts per division of talea rhythm-maker.

Here's a talea:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
... )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```

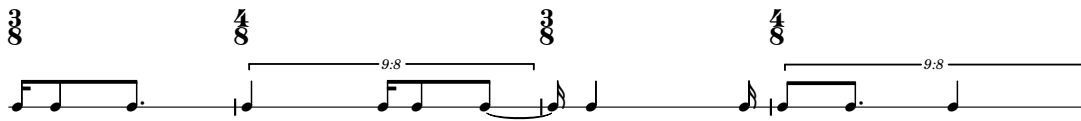


Here's the same rhythm with an extra count added to every other division:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     extra_counts_per_division=(0, 1),
... )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
```

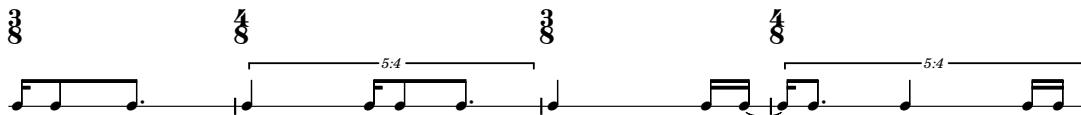
```
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



And here's the same rhythm with two extra counts added to every other division:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     extra_counts_per_division=(0, 2,),
... )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Note that the duration of each added count is equal to the duration of each count in the rhythm-maker's input talea.

Returns integer tuple or none.

TaleaRhythmMaker.helper_functions

Gets helper functions of talea rhythm-maker.

Returns dictionary or none.

TaleaRhythmMaker.split_divisions_by_counts

Gets secondary divisions of talea rhythm-maker.

Secondary divisions impose a cyclic split operation on divisions.

Here's a talea equal to two thirty-second repeating indefinitely. The maker makes four divisions equal to 12 thirty-second notes each:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(2,),
...     denominator=32,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
... )
```

```
>>> divisions = [(3, 8), (3, 8), (3, 8), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```

3
8



Here's the same talea with secondary divisions set to split the divisions every 17 thirty-second notes. The maker makes six divisions with durations equal, respectively, to 12, 5, 7, 10, 2 and 12 thirty-second notes.

Note that $12 + 5 = 17$ and $7 + 10 = 17$:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(2,),
...     denominator=32,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     split_divisions_by_counts=(17,)
... )

>>> divisions = [(3, 8), (3, 8), (3, 8), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```

3
8



Note that the additional divisions created when using *split_divisions_by_counts* are subject to *extra_counts_per_division* just like other divisions.

This example adds one extra thirty-second note to every other division. The durations of the divisions remain the same as in the previous example. But now every other division is tupletted:

```
>>> talea = rhythmmakertools.Talea(
...     counts=(2,),
...     denominator=32,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
...     split_divisions_by_counts=(17,),
...     extra_counts_per_division=(0, 1),
... )

>>> divisions = [(3, 8), (3, 8), (3, 8), (3, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```

3
8



Returns positive integer tuple or none.

`TaleaRhythmMaker.talea`

Gets talea of talea rhythm-maker.

Returns tuple.

`(RhythmMaker).tie_specifier`

Gets tie specifier of rhythm-maker.

Return tie specifier or none.

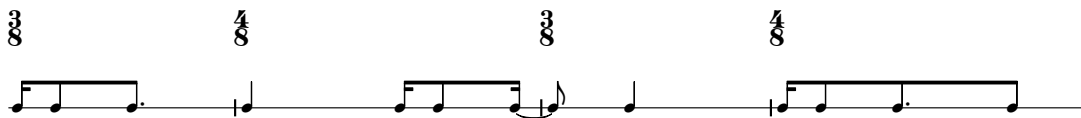
Methods

`TaleaRhythmMaker.reverse()`

Reverses talea rhythm-maker.

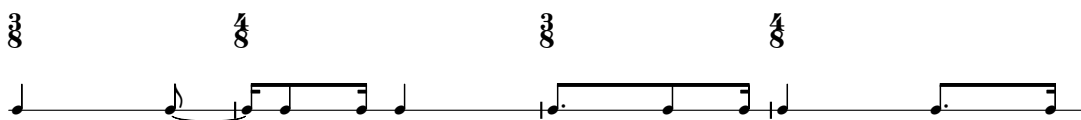
```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
... )
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



```
>>> reversed_maker = maker.reverse()
>>> print(format(reversed_maker))
rhythmmakertools.TaleaRhythmMaker(
    talea=rhythmmakertools.Talea(
        counts=(4, 3, 2, 1),
        denominator=16,
    ),
    burnish_specifier=rhythmmakertools.BurnishSpecifier(
        burnish_divisions=False,
        burnish_output=False,
    ),
    duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(
        decrease_durations_monotonically=False,
    ),
)
```

```
>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> music = reversed_maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Defined equal to copy of this talea rhythm-maker with *talea*, *extra_counts_per_division*, *split_divisions_by_counts*, *burnish_specifier* and *duration_spelling_specifier* reversed.

Returns new talea rhythm-maker.

`TaleaRhythmMaker.rotate(n=0)`

Rotates talea rhythm-maker.

Returns newly constructed rhythm-maker.

Special methods

`TaleaRhythmMaker.__call__` (*divisions*, *seeds=None*)

Calls talea rhythm-maker on *divisions*.

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
... )

>>> divisions = [(3, 8), (4, 8), (3, 8), (4, 8)]
>>> selections = maker(divisions)

>>> for selection in selections:
...     selection
Selection(Note("c'16"), Note("c'8"), Note("c'8."))
Selection(Note("c'4"), Note("c'16"), Note("c'8"), Note("c'16"))
Selection(Note("c'8"), Note("c'4"))
Selection(Note("c'16"), Note("c'8"), Note("c'8."), Note("c'8"))
```

Returns list of of selections.

`(RhythmMaker).__eq__` (*expr*)

Is true when *expr* is a rhythm-maker with type and public properties equal to those of this rhythm-maker. Otherwise false.

Returns boolean.

`TaleaRhythmMaker.__format__` (*format_specification=''*)

Formats talea rhythm-maker.

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

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
... )

>>> print(format(maker))
rhythmmakertools.TaleaRhythmMaker(
    talea=rhythmmakertools.Talea(
        counts=(1, 2, 3, 4),
        denominator=16,
    ),
)
```

Returns string.

`(RhythmMaker).__hash__` ()

Hashes rhythm-maker.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

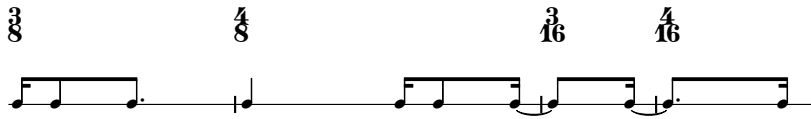
`TaleaRhythmMaker.__illustrate__` (*divisions=None*)

Illustrates talea rhythm-maker.

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
```



```
... )
>>> show(maker)
```



Defaults *divisions* to 3/8, 4/8, 3/16, 4/16.

Returns LilyPond file.

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

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

Returns boolean.

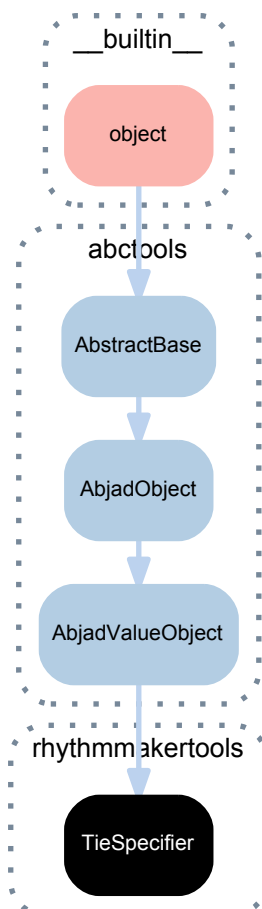
TaleaRhythmMaker **__repr__**()

Gets interpreter representation of talea rhythm-maker.

```
>>> talea = rhythmmakertools.Talea(
...     counts=(1, 2, 3, 4),
...     denominator=16,
... )
>>> rhythmmakertools.TaleaRhythmMaker(
...     talea=talea,
... )
TaleaRhythmMaker(talea=Talea(counts=(1, 2, 3, 4), denominator=16))
```

Returns string.

14.2.14 rhythmmakertools.TieSpecifier



class `rhythmmakertools.TieSpecifier` (*tie_across_divisions=False, tie_split_notes=True*)
Tie specifier.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`TieSpecifier.tie_across_divisions`
Is true when rhythm-maker should tie across divisions. Otherwise false.
Returns boolean.

`TieSpecifier.tie_split_notes`
Is true when rhythm-maker should tie split notes. Otherwise false.
Returns boolean.

Special methods

`TieSpecifier.__eq__` (*arg*)
Is true when *arg* is a tie specifier with values of *tie_across_divisions* and *tie_split_notes* equal to those of this tie specifier. Otherwise false.
Returns boolean.

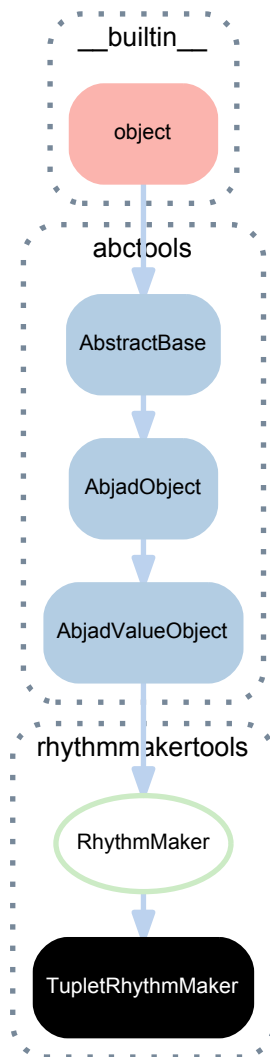
`(AbjadObject).__format__` (*format_specification=''*)
Formats Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

`TieSpecifier.__hash__` ()
Hashes tie specifier.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

14.2.15 `rhythmmakertools.TupletRhythmMaker`



```

class rhythmmakertools.TupletRhythmMaker (
    tuplet_ratios=((1, 1), (1, 2), (1, 3)),
    beam_specifier=None,
    ration_spelling_specifier=None,
    tie_specifier=None,
    tuplet_spelling_specifier=None)
  
```

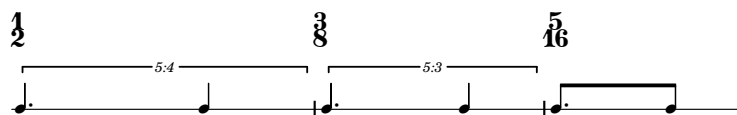
Tuplet rhythm-maker.

Makes tuplets with 3 : 2 leaf ratios:

```

>>> maker = rhythmmakertools.TupletRhythmMaker(
...     tuplet_ratios=[(3, 2)],
... )

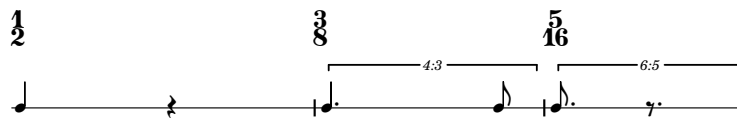
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
  
```



Makes tuplets with alternating 1 : -1 and 3 : 1 leaf ratios:

```
>>> maker = rhythmmakertools.TupletRhythmMaker(
...     tuple_ratios=[(1, -1), (3, 1)],
... )

>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Usage follows the two-step configure-once / call-repeatedly pattern shown here.

Bases

- `rhythmmakertools.RhythmMaker`
- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(`RhythmMaker`).**beam_specifier**
Gets beam specifier of rhythm-maker.

Returns beam specifier or none.

(`RhythmMaker`).**duration_spelling_specifier**
Gets duration spelling specifier of rhythm-maker.

Returns duration spelling specifier or none.

`TupletRhythmMaker`.**tie_specifier**
Gets tie specifier of ratio talea rhythm-maker.

```
>>> tie_specifier = rhythmmakertools.TieSpecifier(
...     tie_across_divisions=True,
... )
>>> maker = rhythmmakertools.TupletRhythmMaker(
...     tuple_ratios=[(2, 3), (1, -2, 1)],
...     tie_specifier=tie_specifier,
... )
```

```
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Returns tie specifier.

`TupletRhythmMaker.tuplet_ratios`

Gets ratio talea of tuplet rhythm-maker.

```
>>> maker = rhythmmakertools.TupletRhythmMaker(
...     tuplet_ratios=[(2, 3), (1, -2, 1)],
...     tie_specifier=tie_specifier,
... )
>>> maker.tuplet_ratios
(Ratio(2, 3), Ratio(1, -2, 1))
```

Returns tuple of ratios.

`TupletRhythmMaker.tuplet_spelling_specifier`

Gets tuplet spelling specifier of tuplet rhythm-maker.

Returns tuplet spelling specifier.

Methods

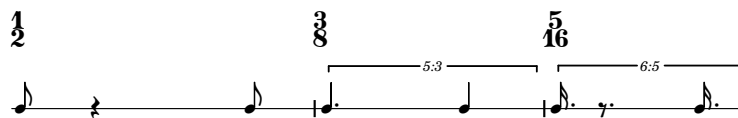
`TupletRhythmMaker.reverse()`

Reverses tuplet rhythm-maker.

```
>>> maker = rhythmmakertools.TupletRhythmMaker(
...     tuplet_ratios=[(2, 3), (1, -2, 1)],
... )
>>> reversed_maker = maker.reverse()
```

```
>>> print(format(reversed_maker))
rhythmmakertools.TupletRhythmMaker(
    tuplet_ratios=(
        mathtools.Ratio(1, -2, 1),
        mathtools.Ratio(3, 2),
    ),
    duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(
        decrease_durations_monotonically=False,
    ),
)
```

```
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> music = reversed_maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```



Defined equal to copy of maker with *tuplet_ratios* and *duration_spelling_specifier* reversed.

Returns new tuplet rhythm-maker.

`TupletRhythmMaker.rotate(n=0)`

Rotates tuplet rhythm-maker.

```
>>> maker = rhythmmakertools.TupletRhythmMaker(
...     tuplet_ratios=[(2, 3), (1, -2, 1), (1, 2, 3)],
... )
>>> rotated_maker = maker.rotate(n=1)
```

```
>>> print(format(rotated_maker))
rhythmmakertools.TupletRhythmMaker(
    tuplet_ratios=(
        mathtools.Ratio(3, 1, 2),
```

```

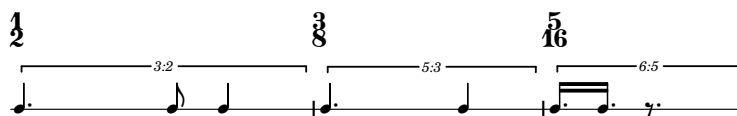
        mathtools.Ratio(3, 2),
        mathtools.Ratio(1, 1, -2),
    ),
)

```

```

>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> music = rotated_maker(divisions)
>>> lilypond_file = rhythm makertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)

```



Returns new tuplet rhythm-maker.

Special methods

`TupletRhythmMaker.__call__(divisions, seeds=None)`

Calls tuplet rhythm-maker on *divisions*.

```

>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> music = maker(divisions)
>>> for division in music:
...     division
Selection(FixedDurationTuplet(Duration(1, 2), "c'4 r4"),)
Selection(FixedDurationTuplet(Duration(3, 8), "c'4. c'8"),)
Selection(FixedDurationTuplet(Duration(5, 16), "c'8. r8."),)

```

Returns list of selections. Each selection holds a single fixed-duration tuplet.

`(RhythmMaker).__eq__(expr)`

Is true when *expr* is a rhythm-maker with type and public properties equal to those of this rhythm-maker. Otherwise false.

Returns boolean.

`TupletRhythmMaker.__format__(format_specification='')`

Formats tuplet rhythm-maker.

```

rhythm makertools.TupletRhythmMaker(
    tuplet_ratios=(
        mathtools.Ratio(1, -1),
        mathtools.Ratio(3, 1),
    ),
)

```

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

Returns string.

`(RhythmMaker).__hash__()`

Hashes rhythm-maker.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(RhythmMaker).__illustrate__(divisions=None)`

Illustrates rhythm-maker.

Defaults *divisions* to 3/8, 4/8, 3/16, 4/16.

Returns LilyPond file.

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

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

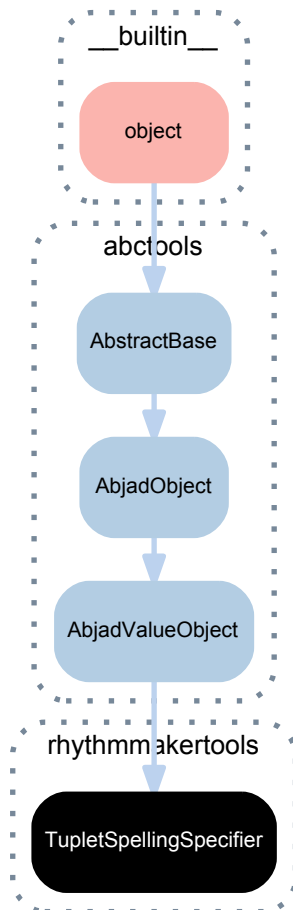
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

14.2.16 `rhythmmakertools.TupletSpellingSpecifier`



class `rhythmmakertools.TupletSpellingSpecifier` (*avoid_dots=False*,
is_diminution=True
 Tuplet spelling specifier.

Bases

- `abctools.AbjadValueObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`TupletSpellingSpecifier.avoid_dots`

Is true when tuplet spelling should avoid dotted rhythmic values. Otherwise false.

Defaults to false.

Returns boolean.

`TupletSpellingSpecifier.is_diminution`

Is true when tuplet should be spelled as diminution. Otherwise false.

Defaults to true.

Returns boolean.

Special methods

`(AbjadValueObject).__eq__(expr)`

Is true when all initialization values of Abjad value object equal the initialization values of *expr*.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`(AbjadValueObject).__hash__()`

Hashes Abjad value object.

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.

14.3 Functions

14.3.1 `rhythmmakertools.make_lilypond_file`

`rhythmmakertools.make_lilypond_file(music, divisions, implicit_scaling=False)`

Makes LilyPond file.

```
>>> maker = rhythmmakertools.EvenRunRhythmMaker(1)
>>> divisions = [(3, 4), (4, 8), (1, 4)]
>>> music = maker(divisions)
>>> lilypond_file = rhythmmakertools.make_lilypond_file(
...     music,
...     divisions,
... )
>>> show(lilypond_file)
```

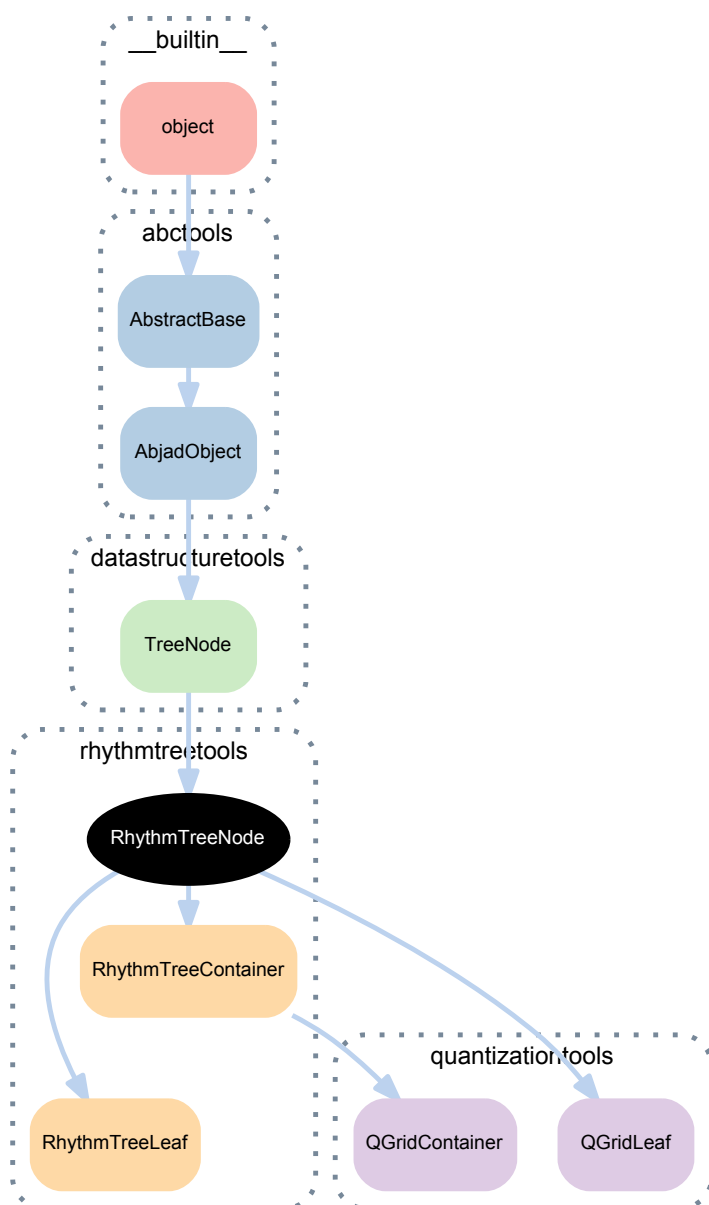


Used in rhythm-maker docs.

Returns LilyPond file.

15.1 Abstract classes

15.1.1 `rhythmtreetools.RhythmTreeNode`



class `rhythmtreetools.RhythmTreeNode` (*preprolated_duration=1, name=None*)
Rhythm-tree node abstract base class.

Bases

- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

Is true when *expr* is a tree node. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

(TreeNode).**__hash__**()

Hashes tree node.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TreeNode).**__ne__**(*expr*)

Is true when tree node does not equal *expr*. Otherwise false.

Returns boolean.

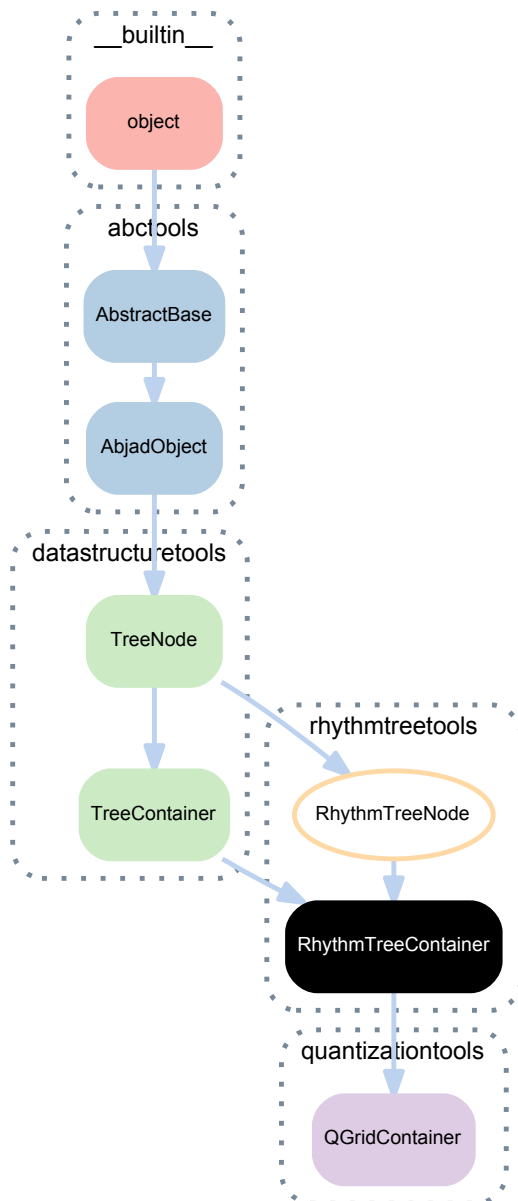
(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

15.2 Concrete classes

15.2.1 `rhythmtreetools.RhythmTreeContainer`



class `rhythmtreetools.RhythmTreeContainer` (*children=None*, *preprolated_duration=1*,
name=None)

A rhythm-tree container.

```
>>> container = rhythmtreetools.RhythmTreeContainer(
...     preprolated_duration=1, children=[])
>>> container
RhythmTreeContainer(
    preprolated_duration=Duration(1, 1)
)
```

Similar to Abjad containers, *RhythmTreeContainer* supports a list interface, and can be appended, extended, indexed and so forth by other *RhythmTreeNode* subclasses:

```
>>> leaf_a = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=1)
>>> leaf_b = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=2)
>>> container.extend([leaf_a, leaf_b])
>>> container
```

```
RhythmTreeContainer(  
    children=(  
        RhythmTreeLeaf(  
            preprolated_duration=Duration(1, 1),  
            is_pitched=True  
        ),  
        RhythmTreeLeaf(  
            preprolated_duration=Duration(2, 1),  
            is_pitched=True  
        ),  
    ),  
    preprolated_duration=Duration(1, 1)  
)
```

```
>>> another_container = rhythmtreetools.RhythmTreeContainer(  
...     preprolated_duration=2)  
>>> another_container.append(  
...     rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3))  
>>> another_container.append(container[1])  
>>> container.append(another_container)  
>>> container  
RhythmTreeContainer(  
    children=(  
        RhythmTreeLeaf(  
            preprolated_duration=Duration(1, 1),  
            is_pitched=True  
        ),  
        RhythmTreeContainer(  
            children=(  
                RhythmTreeLeaf(  
                    preprolated_duration=Duration(3, 1),  
                    is_pitched=True  
                ),  
                RhythmTreeLeaf(  
                    preprolated_duration=Duration(2, 1),  
                    is_pitched=True  
                ),  
            ),  
            preprolated_duration=Duration(2, 1)  
        ),  
    ),  
    preprolated_duration=Duration(1, 1)  
)
```

Call *RhythmTreeContainer* with a *preprolated_duration* to generate a tuplet structure:

```
>>> container((1, 4))  
[FixedDurationTuplet(Duration(1, 4), 'c\'8 FixedDurationTuplet(Duration(1, 4), "c\'8. c\'8")')]
```

Returns *RhythmTreeContainer* instance.

Bases

- `rhythmtreetools.RhythmTreeNode`
- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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**

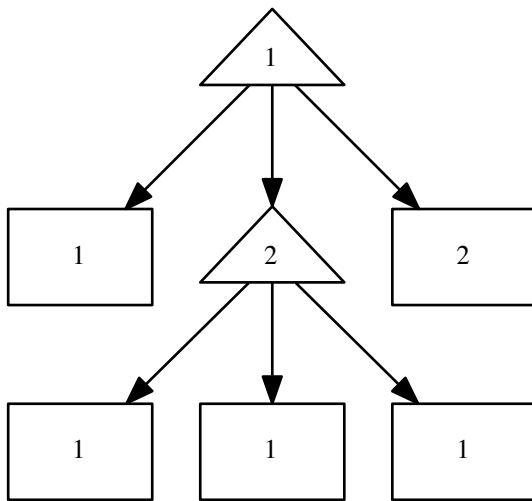
Graphviz format of rhythm tree node.

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 FixedDurationTuplet(Duration(1, 8), "c\'16 c\'16 c\'16") c\'16
```

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 Abjad 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.

RhythmTreeContainer.**__hash__**()

Hashes rhythm-tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

Is 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
```

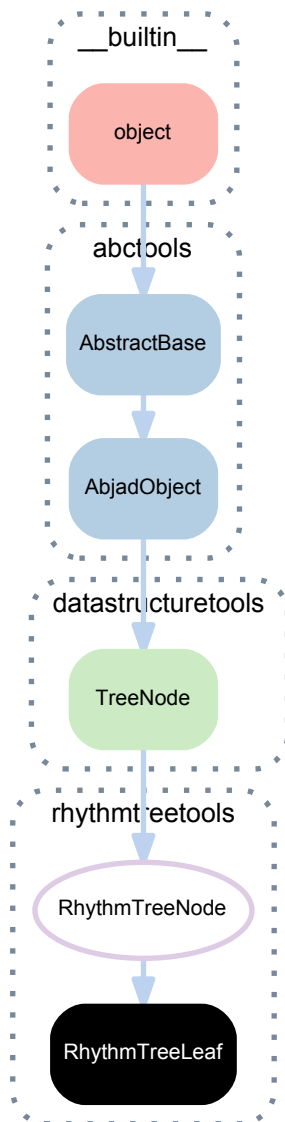
```
>>> c.parent is a
True
```

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

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

Return *None*.

15.2.2 rhythmtreetools.RhythmTreeLeaf



class `rhythmtreetools.RhythmTreeLeaf` (*preprolated_duration=1, is_pitched=True, name=None*)

A rhythm-tree leaf.

```

>>> 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`
- `abctools.AbjadObject.AbstractBase`
- `__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) **.prolotion**

Prolotion 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.

```
>>> rhythmtreetools.RhythmTreeLeaf(1, is_pitched=True).rtm_format
'1'
>>> rhythmtreetools.RhythmTreeLeaf(5, is_pitched=False).rtm_format
'-5'
```

Returns string.

(RhythmTreeNode) **.start_offset**

The starting offset of a node in a rhythm-tree relative the root.

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Returns Offset instance.

(RhythmTreeNode) **.stop_offset**

The stopping offset of a node in a rhythm-tree relative the root.

Read/write properties

RhythmTreeLeaf **.is_pitched**

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*)

Is 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 Abjad object.

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

Returns string.

RhythmTreeLeaf.**__hash__**()

Hashes rhythm-tree leaf.

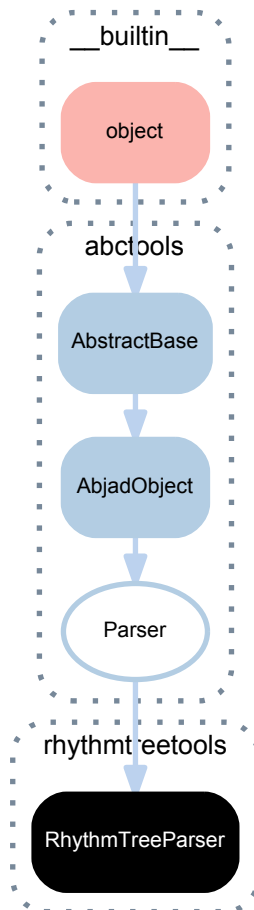
Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

15.2.3 rhythmtreertools.RhythmTreeParser



class rhythmtreertools.**RhythmTreeParser** (*debug=False*)
Rhythm-tree parser.

```

>>> parser = rhythmtreertools.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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

(ObjadObject).__eq__(expr)
    Is true when ID of expr equals ID of Abjad object. Otherwise false.
    Returns boolean.

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

(ObjadObject).__hash__()
    Hashes Abjad object.
    Required to be explicitly re-defined on Python 3 if __eq__ changes.
    Returns integer.

(ObjadObject).__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.

15.3 Functions

15.3.1 `rhythmtreetools.parse_rtm_syntax`

`rhythmtreetools.parse_rtm_syntax(rtm)`

Parses 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:

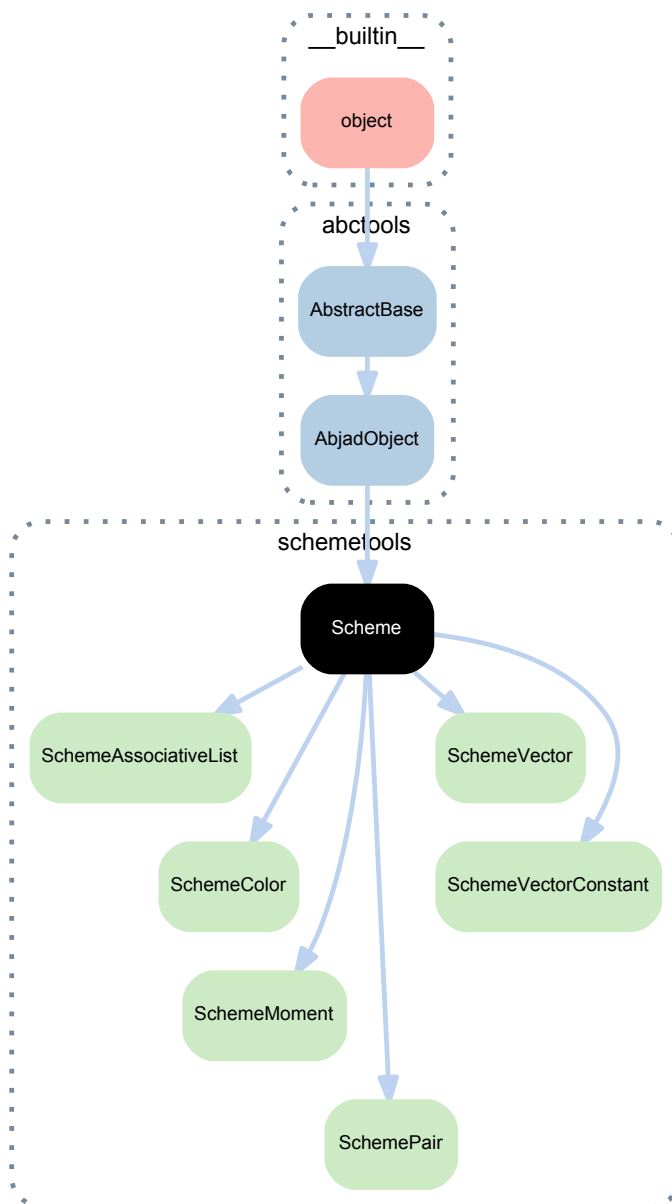
```
>>> rtm = '(3/4 (1 1/2 (4/3 (1 -1/2 1))))'
>>> rhythmtreetools.parse_rtm_syntax(rtm)
FixedDurationTuplet(Duration(3, 16), 'c\'8 c\'16 FixedDurationTuplet(Duration(1, 6), "c\'8 r16 c\'8")')
```

```
>>> print(format(_))
\tweak #'text #tuplet-number::calc-fraction-text
\times 9/17 {
  c'8
  c'16
  \tweak #'edge-height #'(0.7 . 0)
  \times 8/15 {
    c'8
    r16
    c'8
  }
}
```

Returns fixed-duration tuplet or container.

16.1 Concrete classes

16.1.1 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`Scheme.force_quotes`

Is true when quotes should be forced in output. Otherwise false.

Returns boolean.

Static methods

`Scheme.format_embedded_scheme_value` (*value*, *force_quotes=False*)

Formats *value* as an embedded Scheme value.

`Scheme.format_scheme_value` (*value*, *force_quotes=False*)

Formats *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)`

Is 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.

`Scheme.__hash__()`

Hashes scheme.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

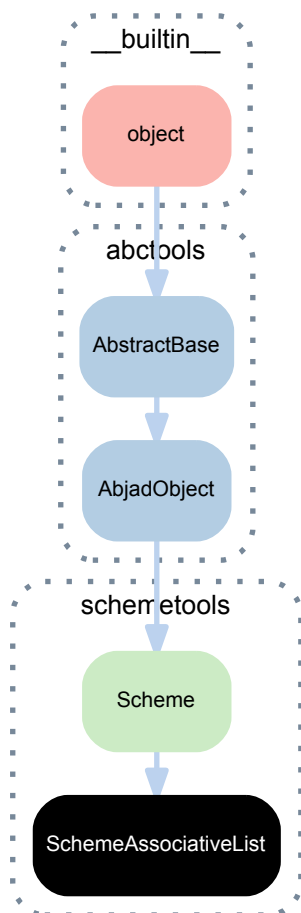
Returns string.

`Scheme.__str__()`

String representation of scheme object.

Returns string.

16.1.2 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Scheme).force_quotes`

Is true when quotes should be forced in output. Otherwise false.

Returns boolean.

Static methods

(Scheme) .**format_embedded_scheme_value** (value, force_quotes=False)
 Formats *value* as an embedded Scheme value.

(Scheme) .**format_scheme_value** (value, force_quotes=False)
 Formats *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)
 Is 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.

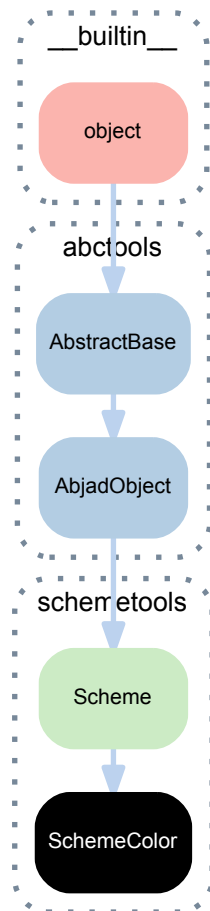
(Scheme) .**__hash__** ()
 Hashes scheme.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 Returns integer.

(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.

`(Scheme).__str__()`
 String representation of scheme object.
 Returns string.

16.1.3 schemetools.SchemeColor



class `schemetools.SchemeColor(*args, **kwargs)`
 A Scheme color.

```
>>> schemetools.SchemeColor('ForestGreen')
SchemeColor('ForestGreen')
```

Scheme colors are immutable.

Bases

- `schemetools.Scheme`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Scheme).force_quotes`
 Is true when quotes should be forced in output. Otherwise false.

Returns boolean.

Static methods

(Scheme) .**format_embedded_scheme_value** (value, force_quotes=False)
 Formats *value* as an embedded Scheme value.

(Scheme) .**format_scheme_value** (value, force_quotes=False)
 Formats *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)
 Is 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.

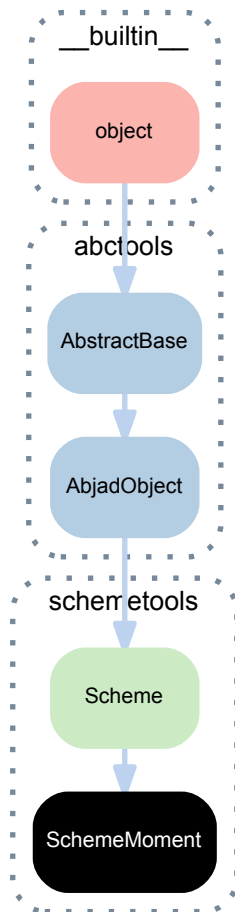
(Scheme) .**__hash__** ()
 Hashes scheme.
 Required to be explicitly re-defined on Python 3 if *__eq__* changes.
 Returns integer.

(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.

`(Scheme).__str__()`
 String representation of scheme object.
 Returns string.

16.1.4 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`
- `abctools.AbjadObject.AbstractBase`
- `__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`

Is true when quotes should be forced in output. Otherwise false.

Returns boolean.

Static methods

`(Scheme).format_embedded_scheme_value(value, force_quotes=False)`

Formats *value* as an embedded Scheme value.

`(Scheme).format_scheme_value(value, force_quotes=False)`

Formats *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)`

Is 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.__hash__()`

Hashes scheme moment.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`SchemeMoment.__le__(other)`

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

`SchemeMoment.__lt__(arg)`

Is 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.

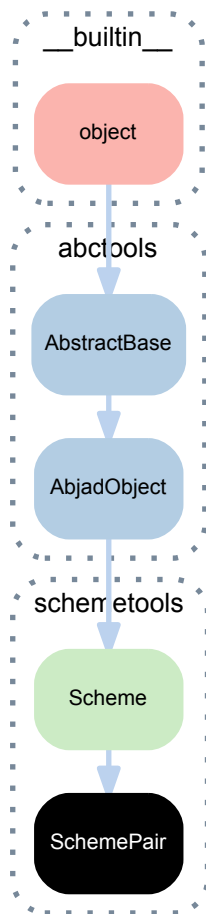
Returns string.

`(Scheme).__str__()`

String representation of scheme object.

Returns string.

16.1.5 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Scheme).force_quotes`

Is true when quotes should be forced in output. Otherwise false.

Returns boolean.

Static methods

(Scheme) .**format_embedded_scheme_value** (*value*, *force_quotes=False*)
Formats *value* as an embedded Scheme value.

(Scheme) .**format_scheme_value** (*value*, *force_quotes=False*)
Formats *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*)
Is 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.

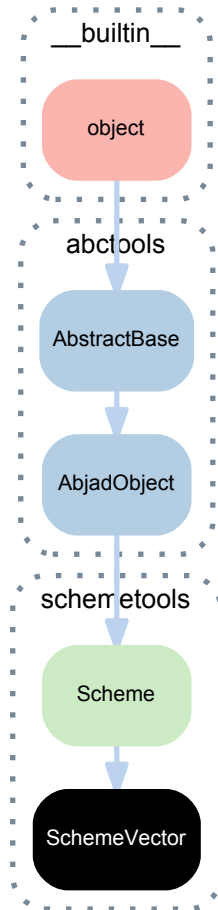
(Scheme) .**__hash__** ()
Hashes scheme.
Required to be explicitly re-defined on Python 3 if *__eq__* changes.
Returns integer.

(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.

`(Scheme).__str__()`
 String representation of scheme object.
 Returns string.

16.1.6 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Scheme) .**force_quotes**

Is true when quotes should be forced in output. Otherwise false.

Returns boolean.

Static methods

(Scheme) .**format_embedded_scheme_value** (*value*, *force_quotes=False*)

Formats *value* as an embedded Scheme value.

(Scheme) .**format_scheme_value** (*value*, *force_quotes=False*)

Formats *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*)

Is 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.

(Scheme) .**__hash__** ()

Hashes scheme.

Required to be explicitly re-defined on Python 3 if *__eq__* changes.

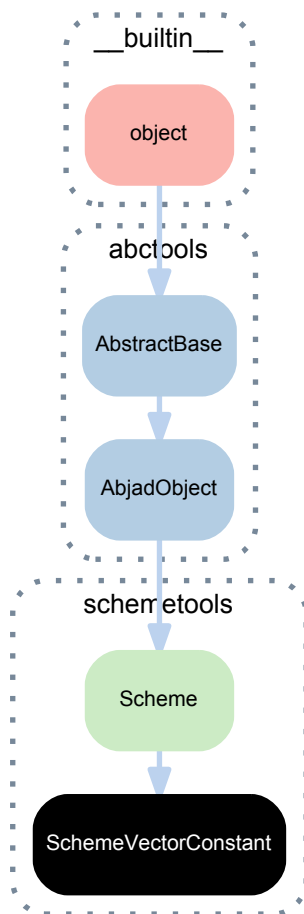
Returns integer.

(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.

(Scheme).**__str__**()
 String representation of scheme object.
 Returns string.

16.1.7 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`

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

Read-only properties

(Scheme) **.force_quotes**

Is true when quotes should be forced in output. Otherwise false.

Returns boolean.

Static methods

(Scheme) **.format_embedded_scheme_value** (*value*, *force_quotes=False*)

Formats *value* as an embedded Scheme value.

(Scheme) **.format_scheme_value** (*value*, *force_quotes=False*)

Formats *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*)

Is 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.

(Scheme) **.__hash__** ()

Hashes scheme.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

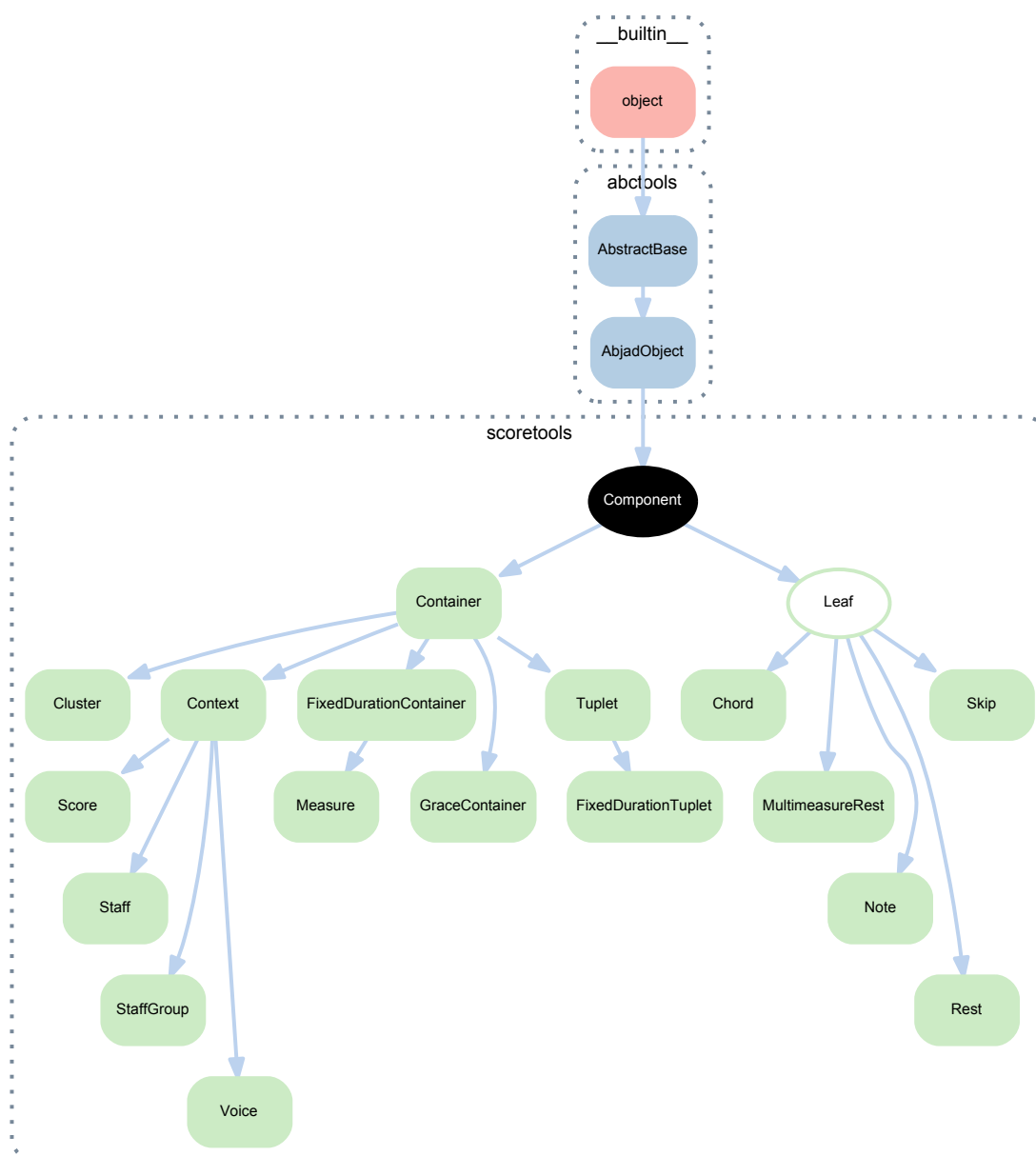
(Scheme) .**__str__**()

String representation of scheme object.

Returns string.

17.1 Abstract classes

17.1.1 scoretools.Component



`class scoretools.Component`

Abstract base class from which score components inherit.

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

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`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.

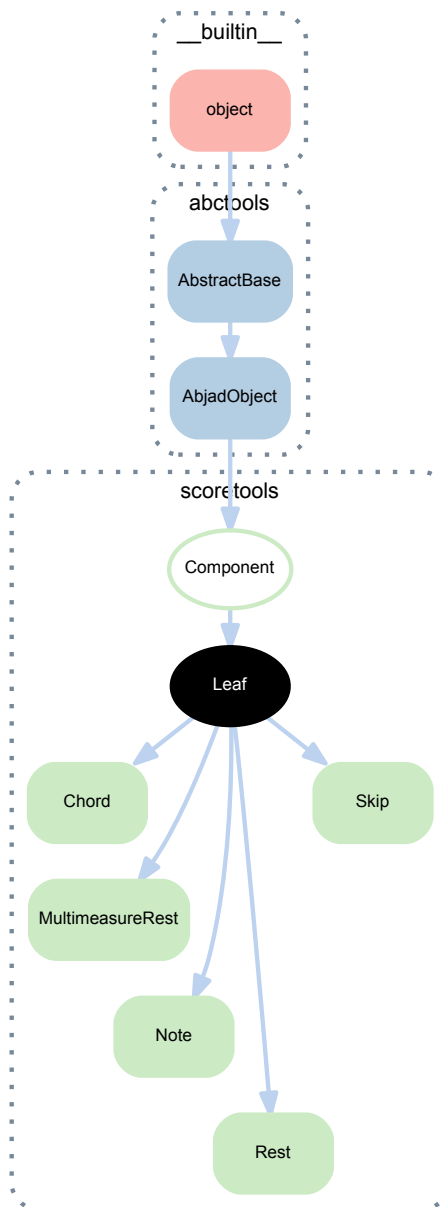
Returns string.

`Component.__rmul__(n)`

Copies component *n* times and detach spanners.

Returns list of new components.

17.1.2 scoretools.Leaf



class `scoretools.Leaf` (*written_duration*)
 Abstract base class from which leaves inherit.
 Leaves include notes, rests, chords and skips.

Bases

- `scoretools.Component`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(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.

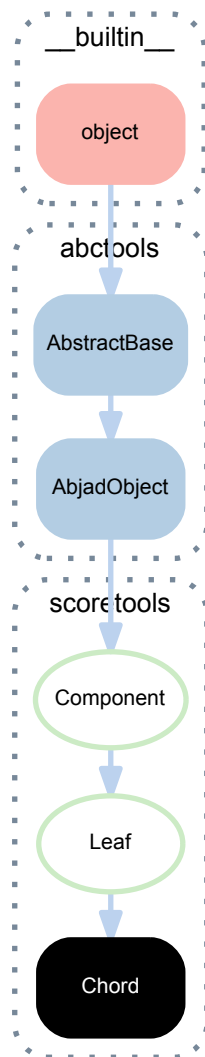
`Leaf.__str__()`

String representation of leaf.

Returns string.

17.2 Concrete classes

17.2.1 scoretools.Chord



```
class scoretools.Chord(*args)
    A chord.
```

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



Bases

- `scoretools.Leaf`
- `scoretools.Component`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)
```



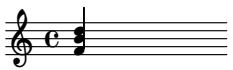
```
>>> for written_pitch in chord.written_pitches:
...     written_pitch
NamedPitch("g' ")
NamedPitch("c'' ")
NamedPitch("e'' ")
```

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.

(AbjadObject) .**__hash__** ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(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.

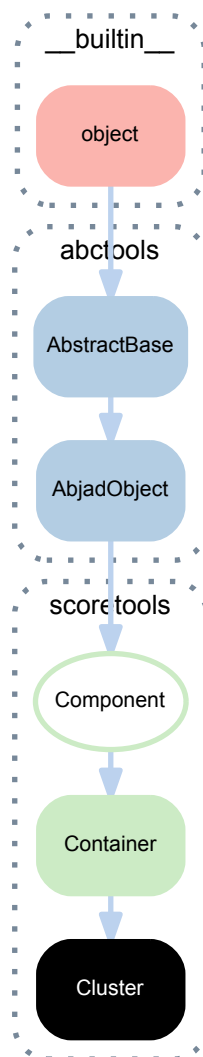
Returns list of new components.

(Leaf) .**__str__** ()

String representation of leaf.

Returns string.

17.2.2 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)
```

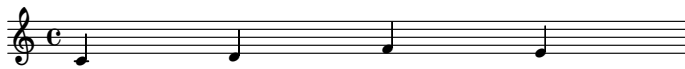


Returns none.

(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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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*)

Is 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.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(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.

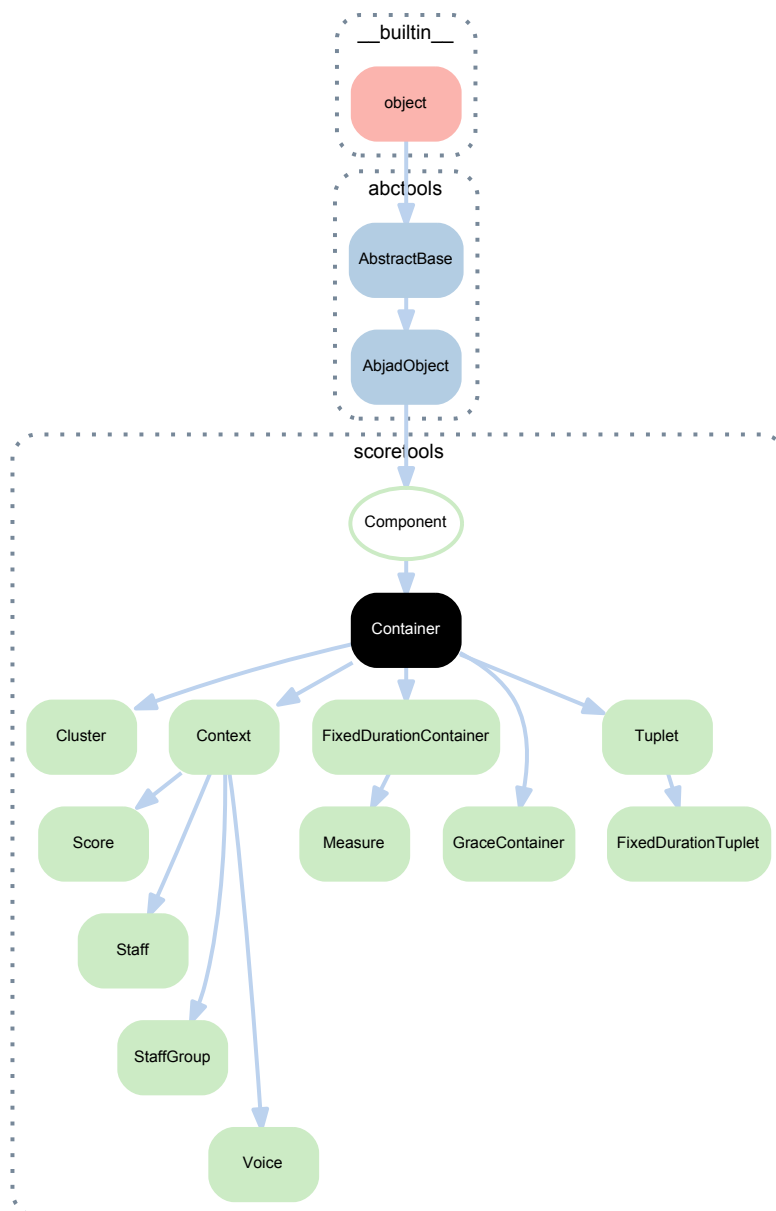
Returns list of new components.

(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.

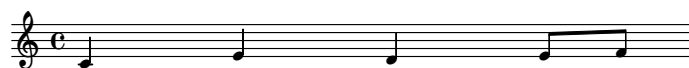
17.2.3 scoretools.Container



class `scoretools.Container` (*music=None, is_simultaneous=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`
- `abctools.AbjadObject.AbstractBase`

- `__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)
```

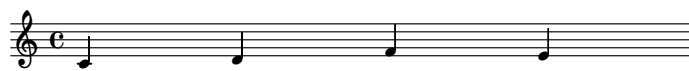


Returns none.

`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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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*)

Is 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.

`(AbjadObject).__hash__` ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(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.

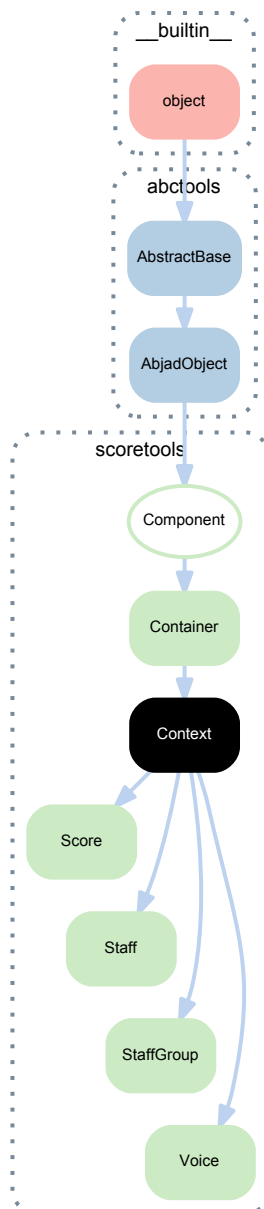
(AbjadObject) .**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.
Returns boolean.

(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.

17.2.4 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
Context()
```

Bases

- `scoretools.Container`
- `scoretools.Component`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`

- `__builtin__.object`

Read-only properties

`Context.consists_commands`

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.consists_commands.append('Horizontal_bracket_engraver')
>>> print(format(staff))
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

`Context.is_semantic`

Is true when context is semantic. Otherwise false.

Returns boolean.

`Context.remove_commands`

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.remove_commands.append('Time_signature_engraver')
>>> print(format(staff))
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

Read/write properties

`Context.context_name`

Gets and sets context name of context.

Returns string.

`Context.is_nonsemantic`

Gets and sets nonsemantic voice flag.

```
>>> pairs = [(1, 8), (5, 16), (5, 16)]
>>> measures = scoretools.make_spacer_skip_measures(pairs)
>>> 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)
```

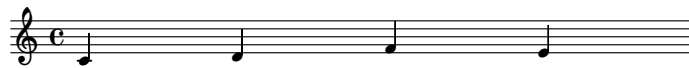


Returns none.

(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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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*)

Is 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.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(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
Context()
```

Returns string.

(Component).**__rmul__**(*n*)

Copies component *n* times and detach spanners.

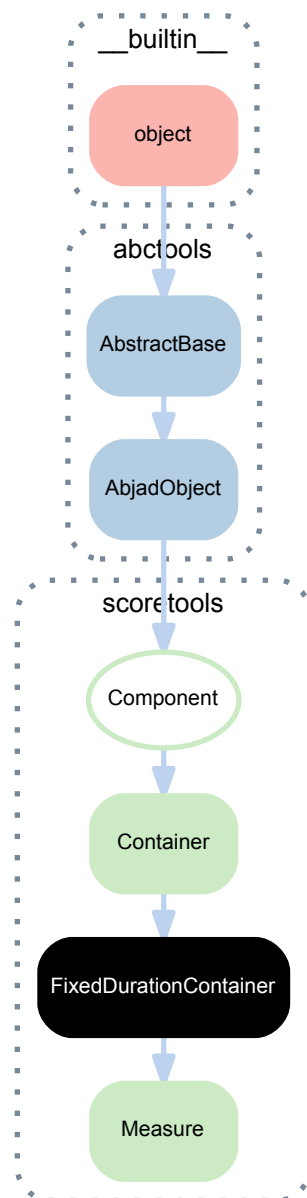
Returns list of new components.

(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.

17.2.5 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`FixedDurationContainer.is_full`

Is true when preprolated duration equals target duration. Otherwise false.

Returns boolean.

`FixedDurationContainer.is_misfilled`

Is true when preprolated duration does not equal target duration. Otherwise false.

Returns boolean.

`FixedDurationContainer.is_overfull`

Is true when preprolated duration is greater than target duration. Otherwise false.

Returns boolean.

`FixedDurationContainer.is_underfull`

Is 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)
```

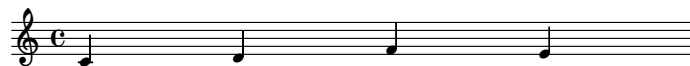


Returns none.

`(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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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*)
Is 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.

(AbjadObject) .**__hash__** ()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 Returns integer.

(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.

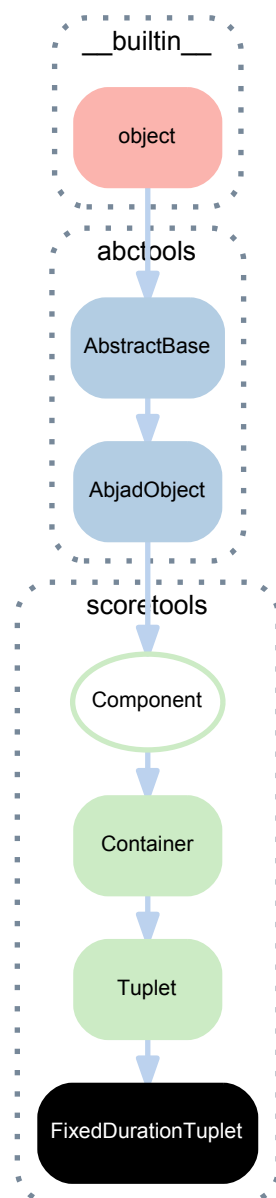
(AbjadObject) .**__ne__** (expr)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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.

17.2.6 scoretools.FixedDurationTuplet



class scoretools.**FixedDurationTuplet** (*duration=Duration(1, 4), music="c'8 c'8 c'8"*)

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

Read-only properties

`(Tuplet).implied_prolation`

Gets implied prolotion 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`

Is true when tuplet multiplier is greater than 1. Otherwise false.

Augmented tuplet:

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
True
```

Diminished tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
False
```

Trivial tuplet:

```
>>> tuplet = Tuplet((1, 1), "c'8. d'8. e'8.")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
False
```

Returns boolean.

(Tuplet).**is_diminution**

Is true when tuplet multiplier is less than 1. Otherwise false.

Augmented tuplet:

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
False
```

Diminished tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
True
```

Trivial tuplet:

```
>>> tuplet = Tuplet((1, 1), "c'8. d'8. e'8.")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
False
```

Returns boolean.

(Tuplet).**is_trivial**

Is 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**

Gets 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**

Gets and sets flag to force fraction formatting of tuplet.

Gets forced fraction formatting of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.force_fraction is None
True
```

Sets forced fraction formatting of tuplet:

```
>>> tuplet.force_fraction = True
>>> show(tuplet)
```



Returns boolean or none.

(Tuplet).**is_invisible**

Gets and sets invisibility status of tuplet.

Gets tuplet invisibility flag:

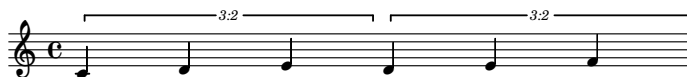
```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



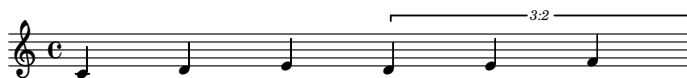
```
>>> tuplet.is_invisible is None
True
```

Sets tuplet invisibility flag:

```
>>> 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

Gets and sets multiplier of fixed-duration tuplet.

```
>>> tuplet = scoretools.FixedDurationTuplet(
...     (1, 4), "c'8 d'8 e'8")
>>> tuplet.multiplier
Multiplier(2, 3)
```

Returns multiplier.

(Tuplet).preferred_denominator

Gets and sets preferred denominator of tuplet.

Gets preferred denominator of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.preferred_denominator is None
True
>>> show(tuplet)
```



Sets 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)
```

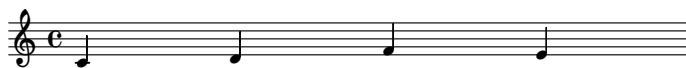


Returns none.

(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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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*.

Sets 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()`

Changes 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()`

Changes 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 prololation 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')`

Trims 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")
```

Preserves fixed-duration tuplet multiplier.

Adjusts 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. Makes augmented tuplet from *duration* and *ratio* and avoid dots.

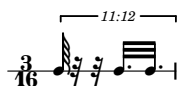
Makes tupletted leaves strictly without dots when all *ratio* equal 1:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



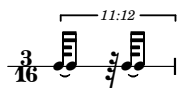
Allows tupletted leaves to return with dots when some *ratio* do not equal 1:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Interprets 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



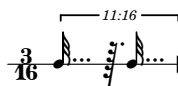
Example 2. Makes augmented tuplet from *duration* and *ratio* and encourages dots:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Interprets nonassignable *ratio* according to *decrease_durations_monotonically*:

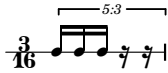
```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=False,
...     decrease_durations_monotonically=False,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 3. Makes diminished tuplet from *duration* and nonzero integer *ratio*.

Makes 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



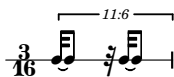
Allows tupletted leaves to return with dots when some *ratio* do not equal 1:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



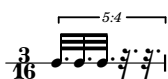
Interprets nonassignable *ratio* according to *decrease_durations_monotonically*:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 4. Makes diminished tuplet from *duration* and *ratio* and encourages 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Interprets 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> 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. Changes leaf to augmented tuplets with *ratio*:

```

>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1),
...     is_diminution=False,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)

```



Example 1b. Changes leaf to augmented tuplets with *ratio*:

```

>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=False,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)

```

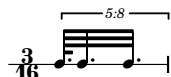


Example 1c. Changes leaf to augmented tuplets with *ratio*:

```

>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2),
...     is_diminution=False,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)

```



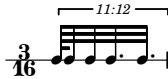
Example 1d. Changes leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 1e. Changes 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])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



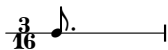
Example 1f. Changes leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2, 3, 3, 4),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 2a. Changes leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```

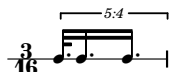


Example 2b. Changes leaf to diminished tuplets with *ratio*:

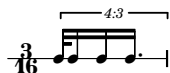
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```

**Example 2c.** Changes leaf to diminished tuplets with *ratio*:

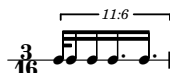
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```

**Example 2d.** Changes leaf to diminished tuplets with *ratio*:

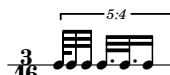
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```

**Example 2e.** Changes 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])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```

**Example 2f.** Changes 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])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Returns tuplet.

(Tuplet).**from_nonreduced_ratio_and_nonreduced_fraction**(*ratio*, *fraction*)
 Makes tuplet from nonreduced *ratio* and nonreduced *fraction*.

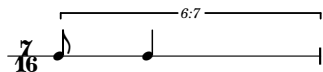
Example 1. Makes container when no prolation is necessary:


```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1),
...     mathtools.NonreducedFraction(7, 16),
...     )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```

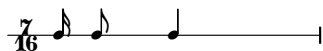


Example 2. Makes fixed-duration tuplet when prolation is necessary:

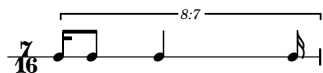
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2),
...     mathtools.NonreducedFraction(7, 16),
...     )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



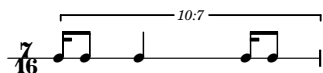
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4),
...     mathtools.NonreducedFraction(7, 16),
...     )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



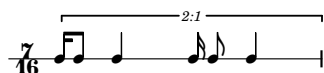
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1),
...     mathtools.NonreducedFraction(7, 16),
...     )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Interprets *d* as tuplet denominator.

Returns tuplet or container.

Special methods

(Container).**__contains__**(*expr*)

Is 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.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(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.

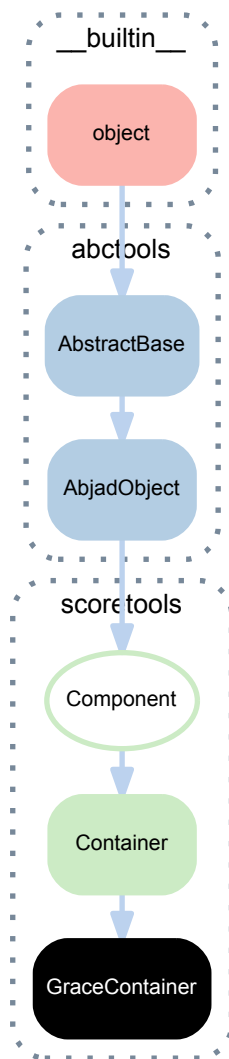
Returns list of new components.

`(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.

17.2.7 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)
```

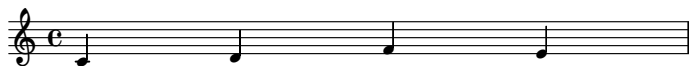


Returns none.

(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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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*)
Is 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.

(AbjadObject).**__hash__**()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
Returns integer.

(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.

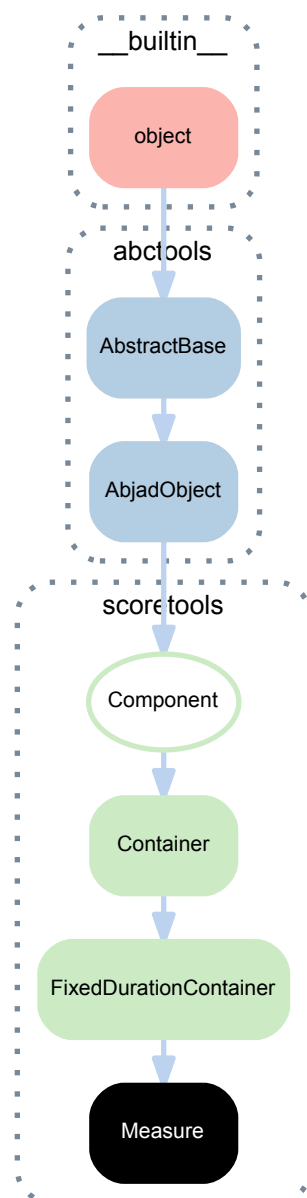
Returns list of new components.

(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.

17.2.8 scoretools.Measure



class `scoretools.Measure` (*time_signature=None, music=None, implicit_scaling=False*)
 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`
- `abctools.AbjadObject.AbstractBase`

- `__builtin__.object`

Read-only properties

`Measure.has_non_power_of_two_denominator`

Is true when measure time signature denominator is not an integer power of 2.

```
>>> measure = Measure((5, 9), "c'8 d' e' f' g'")
>>> measure.implicit_scaling = True
>>> 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`

Is 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'")
>>> measure.implicit_scaling = True
>>> show(measure)
```



```
>>> measure.has_power_of_two_denominator
False
```

Returns boolean.

`Measure.implicit_prolation`

Implied prolation of measure.

Measures with implicit scaling scale the duration of their contents:

```
>>> measure = Measure((5, 12), "c'8 d'8 e'8 f'8 g'8")
>>> measure.implicit_scaling = True
>>> show(measure)
```



```
>>> measure.implicit_prolation
Multiplier(2, 3)
```

```
>>> for note in measure:
...     note, inspect_(note).get_duration()
(Note("c'8"), Duration(1, 12))
(Note("d'8"), Duration(1, 12))
(Note("e'8"), Duration(1, 12))
(Note("f'8"), Duration(1, 12))
(Note("g'8"), Duration(1, 12))
```

Measures without implicit scaling turned on do not scale the duration of their contents:

```
>>> measure = Measure((5, 12), [])
>>> measure.implicit_scaling = False
```

```
>>> measure.implicit_prolation
Multiplier(1, 1)
```

Returns positive multiplier.

Measure.is_full

Is 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

Is 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

Is 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

Is 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

Gets 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

Gets target duration of measure.

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

Gets 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.

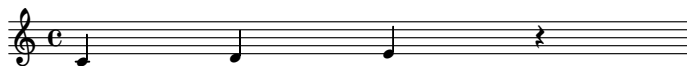
Measure.automatically_adjust_time_signature

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.

`Measure.implicit_scaling`

Is true when measure should scale contents. Otherwise 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)
```



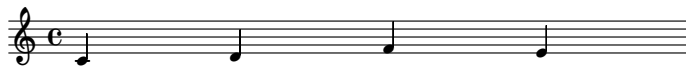
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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)
```

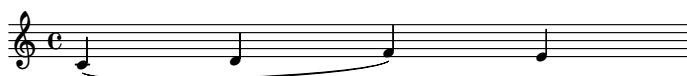


Returns component.

`(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.

Scales measure by non-power-of-two multiplier:

```
>>> measure = Measure((3, 8), "c'8 d'8 e'8")
>>> measure.implicit_scaling = True
>>> 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

(Container) .**__contains__**(*expr*)
Is 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 measure item *i*.

```
>>> 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.

(AbjadObject) .**__hash__**()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
Returns integer.

(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 measure item *i* to *expr*.

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
>>> measure.automatically_adjust_time_signature = True
>>> show(measure)
```

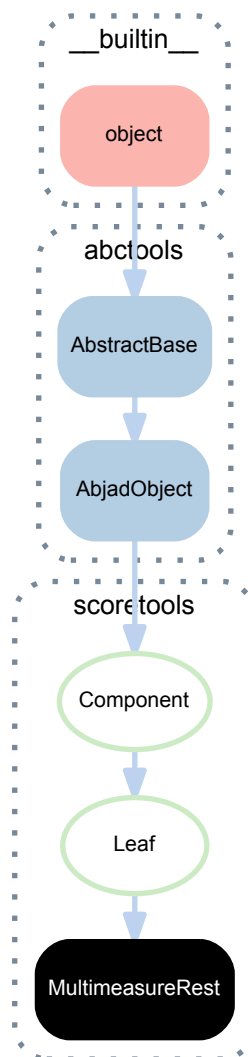


```
>>> measure[1] = Note("ds'8.")
>>> show(measure)
```



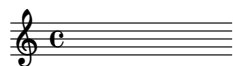
Returns none.

17.2.9 scoretools.MultimeasureRest



```
class scoretools.MultimeasureRest(*args)
    A multimeasure rest.
```

```
>>> rest = scoretools.MultimeasureRest((1, 4))
>>> show(rest)
```



Bases

- `scoretools.Leaf`
- `scoretools.Component`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

(AbjadObject) **.__hash__**()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if **.__eq__** changes.
 Returns integer.

(Component) **.__illustrate__**()
 Illustrates component.
 Returns LilyPond file.

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

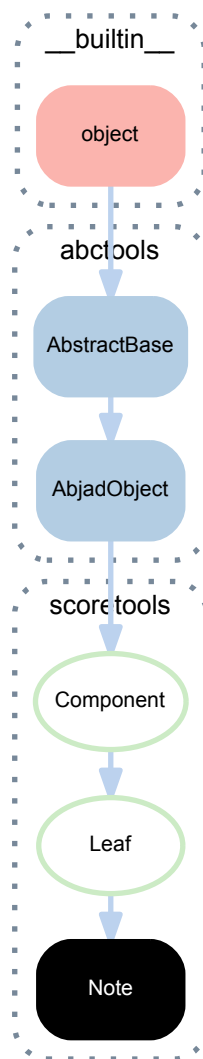
(AbjadObject) **.__ne__**(expr)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(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.

17.2.10 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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

(AbjadObject) .**__hash__** ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(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.

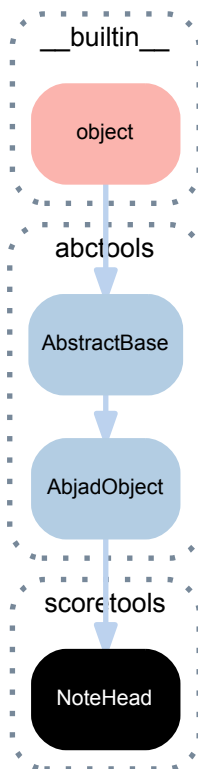
Returns list of new components.

(Leaf) .**__str__** ()

String representation of leaf.

Returns string.

17.2.11 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

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

`NoteHead.__gt__(other)`

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

`NoteHead.__hash__()`

Hashes note-head.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`NoteHead.__le__(other)`

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

`NoteHead.__lt__(expr)`

Is 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.

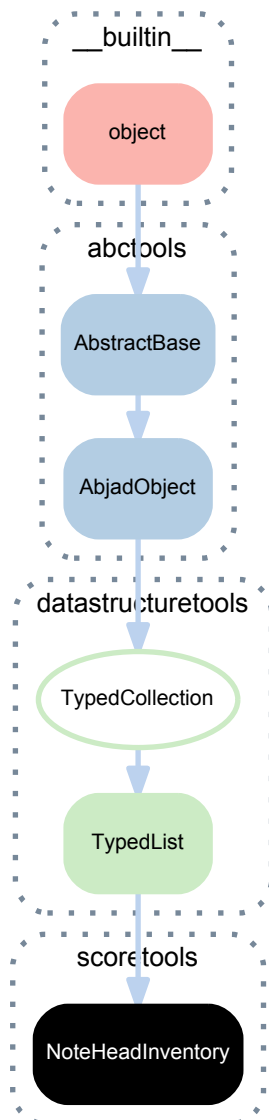
`NoteHead.__str__()`

String representation of note-head.

```
>>> str(note_head)
"cs'")
```

Returns string.

17.2.12 scoretools.NoteHeadInventory



class scoretools.**NoteHeadInventory** (*items=None, client=None*)
 An ordered list of note heads.

```
>>> chord = Chord([0, 1, 4], (1, 4))
>>> inventory = scoretools.NoteHeadInventory(
...     client=chord,
...     items=[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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`NoteHeadInventory.client`
The note head inventory's chord client.

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

```

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

```

```

>>> integer_collection.count(0)
3

```

Returns count.

(`TypedList`) **.extend** (*items*)

Changes *items* 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** (*item*)

Changes *item* to item and returns index.

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

Returns index.

(`TypedList`) **.insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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__** (*item*)

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

(TypedList) **.__delitem__** (*i*)

Aliases list.__delitem__().

Returns none.

(TypedCollection) **.__eq__** (*expr*)

Is 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__().

Returns item.

(TypedCollection).**__hash__**()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TypedList).**__iadd__**(*expr*)

Changes items 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(
            name='ppp',
        ),
        indicatortools.Dynamic(
            name='p',
        ),
        indicatortools.Dynamic(
            name='mp',
        ),
        indicatortools.Dynamic(
            name='mf',
        ),
        indicatortools.Dynamic(
            name='fff',
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__`().

Returns generator.

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

Changes items in *expr* to items and sets.

```
>>> pitch_collection[-1] = 'gqs,'
>>> print(format(pitch_collection))
datastructuretools.TypedList(
    [
        pitchtools.NamedPitch("c"),
```

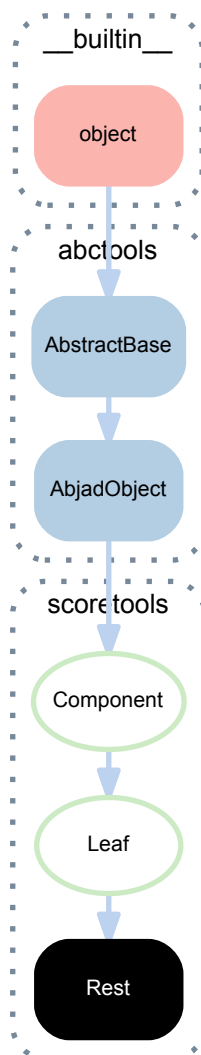


```
pitchtools.NamedPitch("d'"),
pitchtools.NamedPitch("e'"),
pitchtools.NamedPitch('gqs, '),
],
item_class=pitchtools.NamedPitch,
)
```

```
>>> 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,
)
```

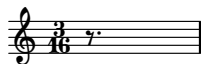
Returns none.

17.2.13 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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

(`AbjadObject`) **.__hash__** ()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 Returns integer.

(`Component`) **.__illustrate__** ()
 Illustrates component.
 Returns LilyPond file.

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

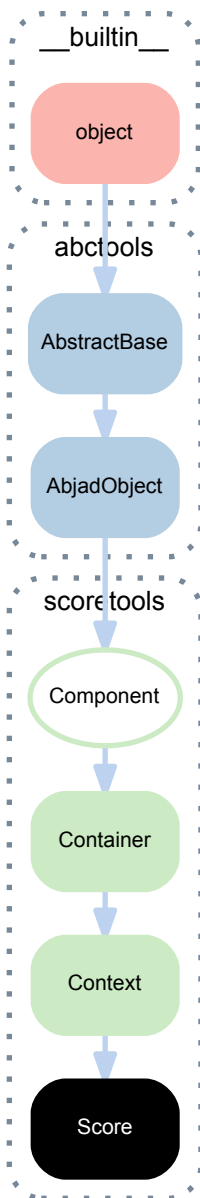
(AbjadObject) .**__ne__**(*expr*)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

(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.

17.2.14 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Context) **.consists_commands**

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.consists_commands.append('Horizontal_bracket_engraver')
>>> print(format(staff))
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

(Context) **.is_semantic**

Is true when context is semantic. Otherwise false.

Returns boolean.

(Context) **.remove_commands**

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.remove_commands.append('Time_signature_engraver')
>>> print(format(staff))
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

Read/write properties

(Context) **.context_name**

Gets and sets context name of context.

Returns string.

(Context) **.is_nonsemantic**

Gets and sets nonsemantic voice flag.

```
>>> pairs = [(1, 8), (5, 16), (5, 16)]
>>> measures = scoretools.make_spacer_skip_measures(pairs)
>>> 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.

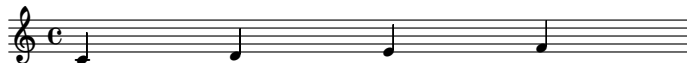
Methods

`Score.add_final_bar_line(abbreviation='.')`

Add final bar line to end of score.

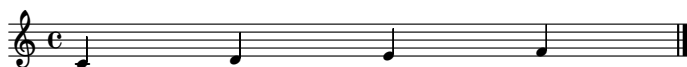
```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> score = Score([staff])
```

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



```
>>> score.add_final_bar_line()
BarLine('|.')
```

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



Returns bar line.

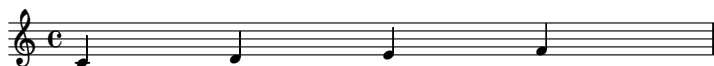
`Score.add_final_markup(markup, extra_offset=None)`

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(
  contents=(
    markuptools.MarkupCommand(
      'italic',
      markuptools.MarkupCommand(
        'right-column',
        ['Bremen - Boston - LA.', 'Jul 2010 - May 2011.']
      )
    ),
  ),
  direction=Down,
)
```

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



Bremen - Boston - LA.
Jul 2010 - May 2011.

Return *markup*.

`(Container).append(component)`

Appends *component* to container.

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



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



Returns none.

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

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



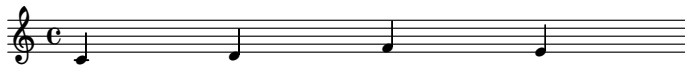
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



Returns none.

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

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



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

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

Returns nonnegative integer.

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

Example 1. Insert note. Do not fracture spanners:

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

Is 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.

(AbjadObject) .**__hash__**()
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(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
Context()
```

Returns string.

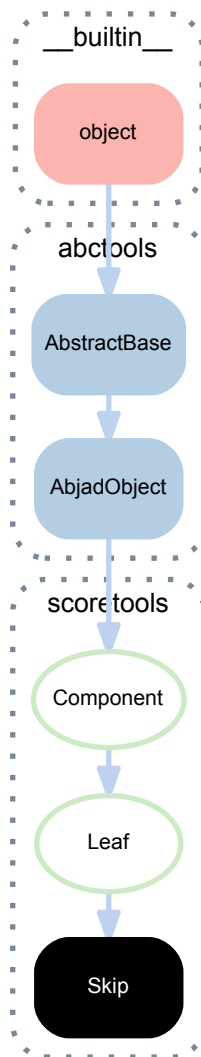
(Component) .**__rmul__**(*n*)
Copies component *n* times and detach spanners.

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.

17.2.15 scoretools.Skip



class `scoretools.Skip(*args)`
 A LilyPond skip.

```
>>> skip = scoretools.Skip((3, 16))
>>> skip
Skip('s8.')
```

Bases

- `scoretools.Leaf`
- `scoretools.Component`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(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.

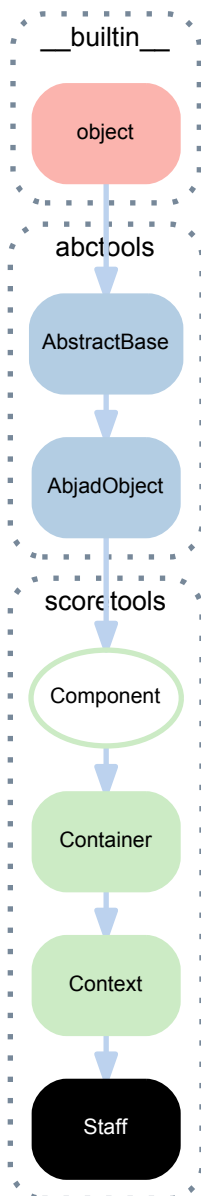
Returns list of new components.

(Leaf).**__str__**()

String representation of leaf.

Returns string.

17.2.16 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Context).consists_commands`

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.consists_commands.append('Horizontal_bracket_engraver')
>>> print(format(staff))
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

`(Context).is_semantic`

Is true when context is semantic. Otherwise false.

Returns boolean.

`(Context).remove_commands`

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.remove_commands.append('Time_signature_engraver')
>>> print(format(staff))
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

Read/write properties

`(Context).context_name`

Gets and sets context name of context.

Returns string.

`(Context).is_nonsemantic`

Gets and sets nonsemantic voice flag.

```
>>> pairs = [(1, 8), (5, 16), (5, 16)]
>>> measures = scoretools.make_spacer_skip_measures(pairs)
>>> 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)
```



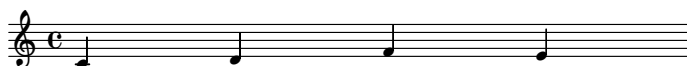
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> show(container)
```



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

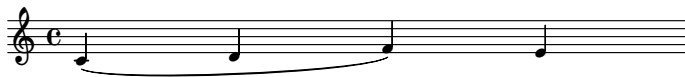


Returns none.

(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*)

Is 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.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(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
Context()
```

Returns string.

(Component).**__rmul__**(*n*)

Copies component *n* times and detach spanners.

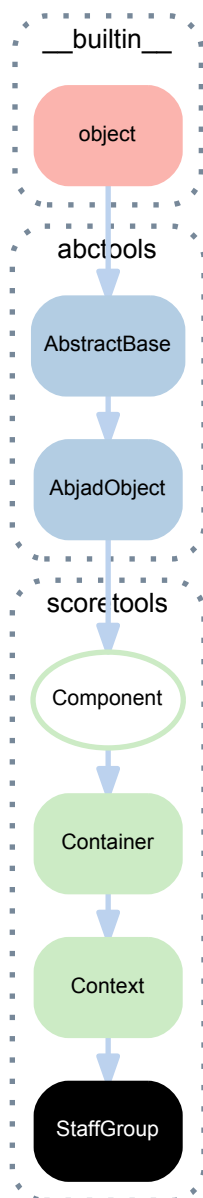
Returns list of new components.

(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.

17.2.17 scoretools.StaffGroup



class `scoretools.StaffGroup` (*music=None, context_name='StaffGroup', name=None*)
 A 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])
```

Bases

- `scoretools.Context`
- `scoretools.Container`
- `scoretools.Component`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`

- `__builtin__.object`

Read-only properties

`(Context).consists_commands`

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.consists_commands.append('Horizontal_bracket_engraver')
>>> print(format(staff))
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

`(Context).is_semantic`

Is true when context is semantic. Otherwise false.

Returns boolean.

`(Context).remove_commands`

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.remove_commands.append('Time_signature_engraver')
>>> print(format(staff))
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

Read/write properties

`(Context).context_name`

Gets and sets context name of context.

Returns string.

`(Context).is_nonsemantic`

Gets and sets nonsemantic voice flag.

```
>>> pairs = [(1, 8), (5, 16), (5, 16)]
>>> measures = scoretools.make_spacer_skip_measures(pairs)
>>> 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.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*)

Is 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.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(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
Context()
```

Returns string.

(Component) .**__rmul__**(*n*)

Copies component *n* times and detach spanners.

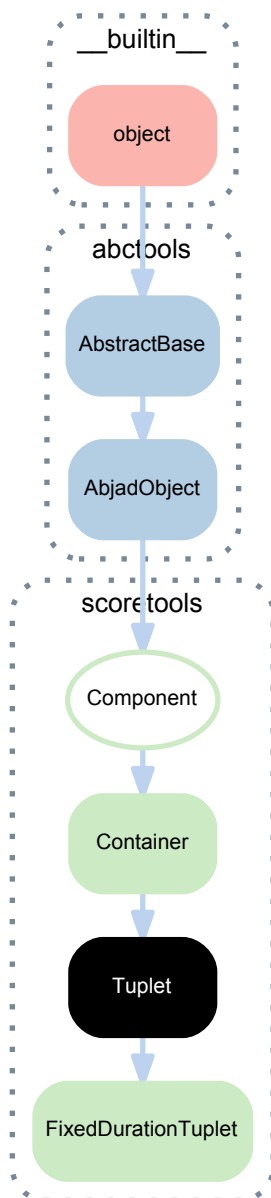
Returns list of new components.

(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.

17.2.18 scoretools.Tuplet



class `scoretools.Tuplet` (*multiplier=None, music=None*)

A tuplet.

A tuplet:

```
>>> tuplet = Tuplet(Multiplier(2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



A nested tuplet:

```
>>> second_tuplet = Tuplet((4, 7), "g'4. ( a'16 )")
>>> tuplet.insert(1, second_tuplet)
>>> show(tuplet)
```



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

Read-only properties

`Tuplet.implied_prolation`

Gets 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`

Is true when tuplet multiplier is greater than 1. Otherwise false.

Augmented tuplet:

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
True
```

Diminished tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
False
```

Trivial tuplet:

```
>>> tuplet = Tuplet((1, 1), "c'8. d'8. e'8.")
>>> show(tuplet)
```



```
>>> tuplet.is_augmentation
False
```

Returns boolean.

Tuplet.is_diminution

Is true when tuplet multiplier is less than 1. Otherwise false.

Augmented tuplet:

```
>>> tuplet = Tuplet((4, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
False
```

Diminished tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'4 d'4 e'4")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
True
```

Trivial tuplet:

```
>>> tuplet = Tuplet((1, 1), "c'8. d'8. e'8.")
>>> show(tuplet)
```



```
>>> tuplet.is_diminution
False
```

Returns boolean.

Tuplet.is_trivial

Is 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`

Gets and sets flag to force fraction formatting of tuplet.

Gets forced fraction formatting of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.force_fraction is None
True
```

Sets forced fraction formatting of tuplet:

```
>>> tuplet.force_fraction = True
>>> show(tuplet)
```



Returns boolean or none.

`Tuplet.is_invisible`

Gets and sets invisibility status of tuplet.

Gets tuplet invisibility flag:

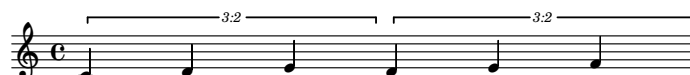
```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



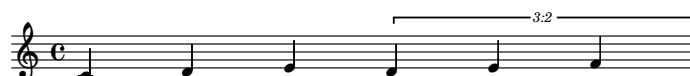
```
>>> tuplet.is_invisible is None
True
```

Sets tuplet invisibility flag:

```
>>> 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**
Gets and sets multiplier of tuplet.

Gets tuplet multiplier:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> show(tuplet)
```



```
>>> tuplet.multiplier
Multiplier(2, 3)
```

Sets tuplet multiplier:

```
>>> tuplet.multiplier = Multiplier(4, 3)
>>> show(tuplet)
```



Returns multiplier.

Tuplet.**preferred_denominator**
Gets and sets preferred denominator of tuplet.

Gets preferred denominator of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.preferred_denominator is None
True
>>> show(tuplet)
```



Sets 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)
```

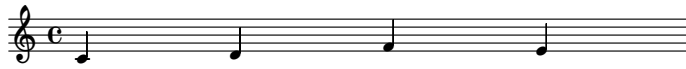


Returns none.

(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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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*.

Sets 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**()

Changes 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. Changes 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. Changes 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. Makes augmented tuplet from *duration* and *ratio* and avoid dots.

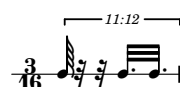
Makes tupletted leaves strictly without dots when all *ratio* equal 1:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



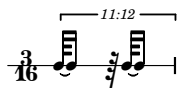
Allows tupletted leaves to return with dots when some *ratio* do not equal 1:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Interprets 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



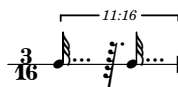
Example 2. Makes augmented tuplet from *duration* and *ratio* and encourages dots:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Interprets nonassignable *ratio* according to *decrease_durations_monotonically*:

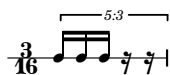
```
>>> tuplet = Tuplet.from_duration_and_ratio(
...     Duration(3, 16),
...     mathtools.Ratio(5, -1, 5),
...     avoid_dots=False,
...     decrease_durations_monotonically=False,
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 3. Makes diminished tuplet from *duration* and nonzero integer *ratio*.

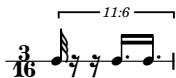
Makes tupletted leaves strictly without dots when all *ratio* equal 1:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



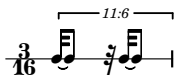
Allows tupletted leaves to return with dots when some *ratio* do not equal 1:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



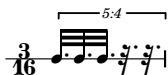
Interprets nonassignable *ratio* according to *decrease_durations_monotonically*:

```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 4. Makes diminished tuplet from *duration* and *ratio* and encourages 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Interprets 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(measure)
```



Reduces *ratio* relative to each other.

Interprets negative *ratio* as rests.

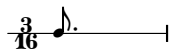
Returns fixed-duration tuplet.

`Tuplet.from_leaf_and_ratio(leaf, ratio, is_diminution=True)`
 Makes tuplet from *leaf* and *ratio*.

```
>>> note = Note("c'8.")
```

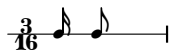
Example 1a. Changes leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1),
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



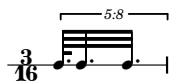
Example 1b. Changes leaf to augmented tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 1c. Changes 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])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 1d. Changes leaf to augmented tuplets with *ratio*:

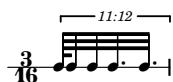
```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=False,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 1e. Changes 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])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



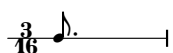
Example 1f. Changes leaf to augmented triplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     mathtools.Ratio(1, 2, 2, 3, 3, 4),
...     is_diminution=False,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 2a. Changes leaf to diminished triplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



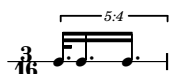
Example 2b. Changes leaf to diminished triplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 2c. Changes leaf to diminished triplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2],
...     is_diminution=True,
...     )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



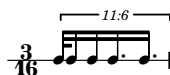
Example 2d. Changes leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



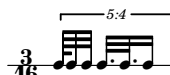
Example 2e. Changes leaf to diminished tuplets with *ratio*:

```
>>> tuplet = Tuplet.from_leaf_and_ratio(
...     note,
...     [1, 2, 2, 3, 3],
...     is_diminution=True,
... )
>>> measure = Measure((3, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Example 2f. Changes 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])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



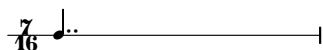
Returns tuplet.

`Tuplet.from_nonreduced_ratio_and_nonreduced_fraction` (*ratio*, *fraction*)

Makes tuplet from nonreduced *ratio* and nonreduced *fraction*.

Example 1. Makes container when no prolation is necessary:

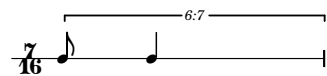
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1),
...     mathtools.NonreducedFraction(7, 16),
... )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



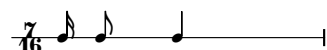
Example 2. Makes fixed-duration tuplet when prolation is necessary:

```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2),
...     mathtools.NonreducedFraction(7, 16),
... )
```

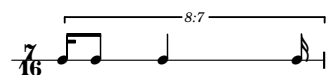
```
>>> measure = Measure((7, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



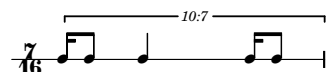
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4),
...     mathtools.NonreducedFraction(7, 16),
...     )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



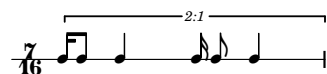
```
>>> tuplet = Tuplet.from_nonreduced_ratio_and_nonreduced_fraction(
...     mathtools.NonreducedRatio(1, 2, 4, 1),
...     mathtools.NonreducedFraction(7, 16),
...     )
>>> measure = Measure((7, 16), [tuplet])
>>> staff = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



```
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> 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 = Staff([measure])
>>> staff.context_name = 'RhythmicStaff'
>>> show(staff)
```



Interprets d as tuplet denominator.

Returns tuplet or container.

Special methods

(Container).**__contains__**(*expr*)

Is 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.

(AbjadObject) .**__hash__** ()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 Returns integer.

(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.

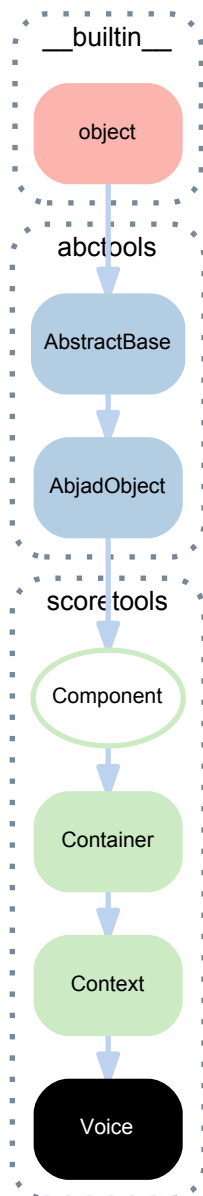
(AbjadObject) .**__ne__** (expr)
 Is true when Abjad object does not equal *expr*. Otherwise false.
 Returns boolean.

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.

17.2.19 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("c'8 d'8 e'8 f'8")
```

```
>>> show(voice)
```



Returns voice instance.

Bases

- `scoretools.Context`

- `scoretools.Container`
- `scoretools.Component`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Context).consists_commands`

Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.consists_commands.append('Horizontal_bracket_engraver')
>>> print(format(staff))
\new Staff \with {
  \consists Horizontal_bracket_engraver
} {
}
```

`(Context).is_semantic`

Is true when context is semantic. Otherwise false.

Returns boolean.

`(Context).remove_commands`

Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.remove_commands.append('Time_signature_engraver')
>>> print(format(staff))
\new Staff \with {
  \remove Time_signature_engraver
} {
}
```

Read/write properties

`(Context).context_name`

Gets and sets context name of context.

Returns string.

`(Context).is_nonsemantic`

Gets and sets nonsemantic voice flag.

```
>>> pairs = [(1, 8), (5, 16), (5, 16)]
>>> measures = scoretools.make_spacer_skip_measures(pairs)
>>> 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)
```



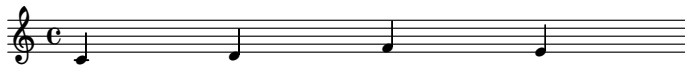
```
>>> notes = [Note("e'32"), Note("d'32"), Note("e'16")]
>>> container.extend(notes)
>>> show(container)
```



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(direction=Down)
>>> attach(slur, container[:])
>>> 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(direction=Down)
>>> attach(slur, container[:])
>>> 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)

Is 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.

(AbjadObject).**__hash__**()
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(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
Context()
```

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.

17.3 Functions

17.3.1 `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")
>>> measure.implicit_scaling = True
>>> 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", implicit_scaling=True)
```

Append nothing to nonunderfull *measure*.

Return *measure*.

17.3.2 `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.

17.3.3 `scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr`

`scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr` (*expr*,

*sup-
ple-
ment=None*)

Applies full-measure tuplets to contents of measures in *expr*:

```
>>> staff = Staff([
...     Measure((2, 8), "c'8 d'8"),
...     Measure((3, 8), "e'8 f'8 g'8")])
>>> show(staff)
```



```
>>> scoretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr(staff)
>>> show(staff)
```



Returns none.

17.3.4 `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.

17.3.5 scoretools.fill_measures_in_expr_with_full_measure_spacer_skips

`scoretools.fill_measures_in_expr_with_full_measure_spacer_skips` (*expr*, *iterctrl=None*)

Fill measures in *expr* with full-measure spacer skips.

17.3.6 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*, *iterctrl=None*)

Fills measures in *expr* with minimal number of notes that decrease durations monotonically:

```
>>> measure = Measure((5, 18), [], implicit_scaling=True)
```

```
>>> 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), [])
>>> measure.implicit_scaling = True
```

```
>>> scoretools.fill_measures_in_expr_with_minimal_number_of_notes(
...     measure, decrease_durations_monotonically=False)
```

Returns none.

17.3.7 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.

17.3.8 scoretools.fill_measures_in_expr_with_time_signature_denominator_notes

`scoretools.fill_measures_in_expr_with_time_signature_denominator_notes` (*expr*, *iterctrl=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.

17.3.9 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.

17.3.10 `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.

17.3.11 `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.

17.3.12 `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.

17.3.13 `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.

17.3.14 `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.

17.3.15 `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 = Staff(leaves)
>>> staff.context_name = 'RhythmicStaff'
>>> 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)
```



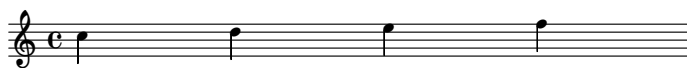
Example 5. Read *pitches* cyclically when the length of *pitches* is less than the length of *durations*:

```
>>> pitches = ['C5']
>>> durations = 2 * [Duration(3, 8), Duration(1, 8)]
>>> leaves = scoretools.make_leaves(pitches, durations)
>>> staff = Staff(leaves)
>>> show(staff)
```



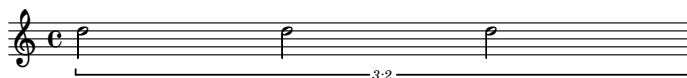
Example 6. Read *durations* cyclically when the length of *durations* is less than the length of *pitches*:

```
>>> 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)
```



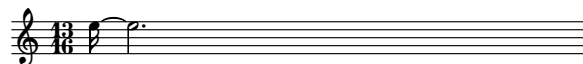
Example 8. Set *decrease_durations_monotonically* to true to return nonassignable durations tied from greatest to least:

```
>>> 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 = Staff(leaves)
>>> staff.context_name = 'RhythmicStaff'
>>> 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)
```

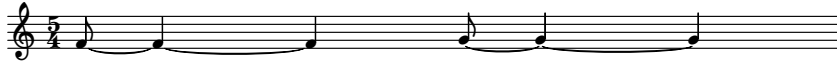


Example 12. You may set *forbidden_written_duration* and *decrease_durations_monotonically* together:

```
>>> 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.

17.3.16 scoretools.make_leaves_from_talea

```
scoretools.make_leaves_from_talea(talea, talea_denominator, de-  
                                crease_durations_monotonically=True,  
                                tie_rests=False, forbidden_written_duration=None)
```

Make leaves from *talea*.

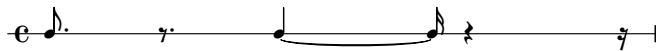
Interpret positive elements in *talea* as notes numerators.

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 = Staff(leaves)
>>> staff.context_name = 'RhythmicStaff'
>>> 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 = Staff(leaves)
>>> staff.context_name = 'RhythmicStaff'
>>> 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 = Staff(leaves)
>>> staff.context_name = 'RhythmicStaff'
>>> time_signature = TimeSignature((4, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



Returns selection.

17.3.17 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.

17.3.18 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.

17.3.19 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 selection.

17.3.20 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.

17.3.21 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)
```



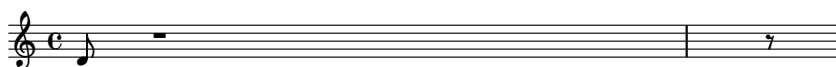
```
>>> leaves = scoretools.make_percussion_note(2, (1, 64), (1, 8))
>>> staff = Staff(leaves)
>>> show(staff)
```



```
>>> 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.

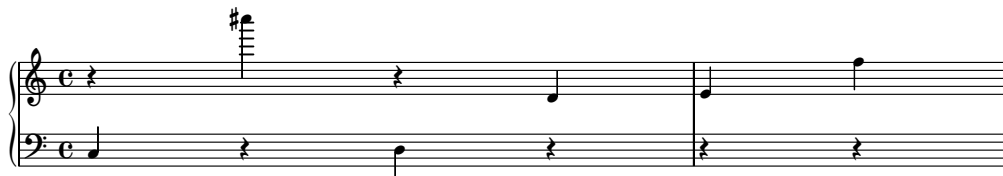
17.3.22 scoretools.make_piano_score_from_leaves

`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.

17.3.23 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.

17.3.24 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.

17.3.25 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.

17.3.26 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.

17.3.27 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)

Makes 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.

17.3.28 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.

17.3.29 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.

17.3.30 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.

17.3.31 scoretools.make_rhythmic_sketch_staff

`scoretools.make_rhythmic_sketch_staff` (*music*)

Make rhythmic staff with transparent time_signature and transparent bar lines.

17.3.32 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.

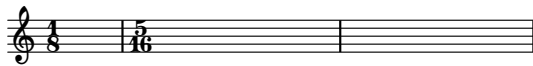
Returns list of skips.

17.3.33 scoretools.make_spacer_skip_measures

`scoretools.make_spacer_skip_measures` (*time_signatures*, *implicit_scaling=False*)

Makes 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.

17.3.34 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)
```



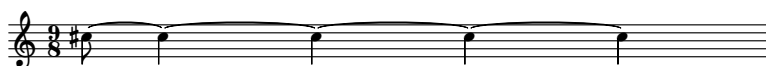
Example 3. Make tied note with half notes forbidden and durations decreasing monotonically:

```
>>> 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.

17.3.35 scoretools.move_full_measure_tuplet_prolation_to_measure_time_signature

`scoretools.move_full_measure_tuplet_prolation_to_measure_time_signature` (*expr*)

Moves prolation of full-measure tuplet to time signature of measure.

Measures usually become non-power-of-two as as result.

```
>>> tuplet = scoretools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> measure = Measure((2, 8), [tuplet])
>>> measure.implicit_scaling = True
>>> scoretools.move_full_measure_tuplet_prolation_to_measure_time_signature(measure)
```

Returns none.

17.3.36 `scoretools.move_measure_prolation_to_full_measure_tuplet`

`scoretools.move_measure_prolation_to_full_measure_tuplet` (*expr*)

Move measure prolotion to full-measure tuplet.

Turn non-power-of-two measures into power-of-two measures containing a single fixed-duration tuplet.

Note that not all non-power-of-two measures can be made power-of-two.

Returns None because processes potentially many measures.

17.3.37 `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")
>>> measure.implicit_scaling = True
>>> 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.", implicit_scaling=True)
>>> 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.

17.3.38 `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())
```

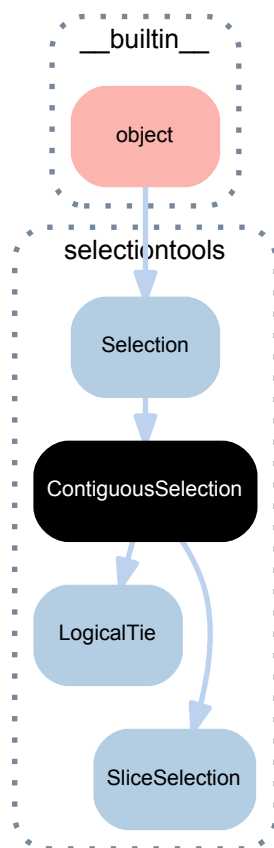
```
>>> scoretools.set_measure_denominator_and_adjust_numerator(measure, 16)
Measure((6, 16), "c'8 d'8 e'8")
```

Leave *measure* contents unchanged.

Return *measure*.

18.1 Concrete classes

18.1.1 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*)
Is true when *expr* is in selection. Otherwise false.
Returns boolean.

(Selection).**__eq__**(*expr*)
Is 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).**__hash__**()
Hashes selection.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
Returns integer.

(Selection).**__illustrate__**()
Attempts to illustrate selection.

Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to 'RhythmicStaff'. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.

The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.

Returns LilyPond file.

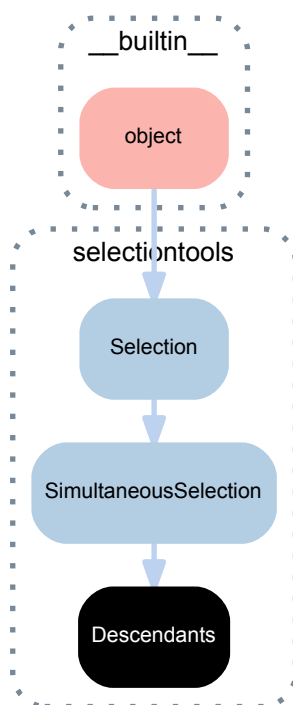
(Selection).**__len__**()
Number of components in selection.
Returns nonnegative integer.

(Selection).**__ne__**(*expr*)
Is 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.

18.1.2 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("c'4")
Note("c'4")
<Staff-"Bass Staff"{1}>
Voice('b,4')
Note('b,4')
```

```
>>> for x in selectiontools.Descendants(score['Bass Voice']): x
...
Voice('b,4')
Note('b,4')
```

Descendants is treated as the selection of the component's improper descendants.

Returns descendants.

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)`

Is true when *expr* is in selection. Otherwise false.

Returns boolean.

`(Selection).__eq__(expr)`

Is 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).__hash__()`

Hashes selection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(Selection).__illustrate__()`

Attempts to illustrate selection.

Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to `'RhythmicStaff'`. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.

The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.

Returns LilyPond file.

(Selection). **__len__**()

Number of components in selection.

Returns nonnegative integer.

(Selection). **__ne__**(*expr*)

Is true when selection does not equal *expr*. Otherwise false.

Returns boolean.

(Selection). **__radd__**(*expr*)

Concatenates selection to *expr*.

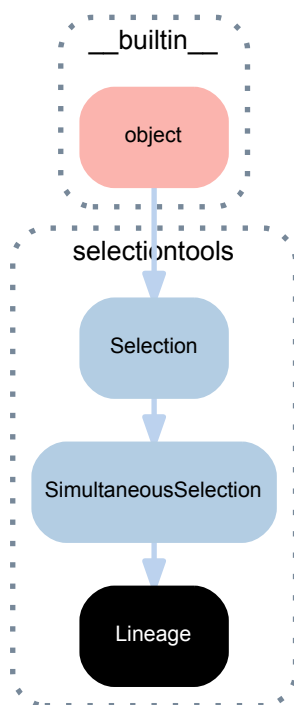
Returns newly created selection.

(Selection). **__repr__**()

Gets interpreter representation of selection.

Returns string.

18.1.3 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("c'4")
Note("c'4")
<Staff-"Bass Staff"{1}>
Voice('b,4')
Note('b,4')
```



```
>>> for x in selectiontools.Lineage(score['Bass Voice']): x
...
<Score<<2>>>
<Staff-"Bass Staff"{1}>
Voice('b,4')
Note('b,4')
```

Returns lineage.

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)`

Is true when *expr* is in selection. Otherwise false.

Returns boolean.

`(Selection).__eq__(expr)`

Is 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) .**__hash__**()

Hashes selection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(Selection) .**__illustrate__**()

Attempts to illustrate selection.

Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to 'RhythmicStaff'. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.

The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.

Returns LilyPond file.

(Selection) .**__len__**()

Number of components in selection.

Returns nonnegative integer.

(Selection) .**__ne__**(*expr*)

Is true when selection does not equal *expr*. Otherwise false.

Returns boolean.

(Selection) .**__radd__**(*expr*)

Concatenates selection to *expr*.

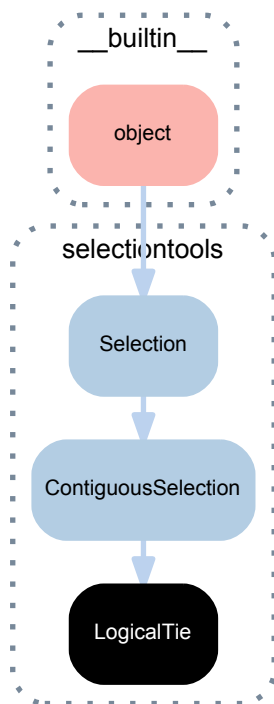
Returns newly created selection.

(Selection) .**__repr__**()

Gets interpreter representation of selection.

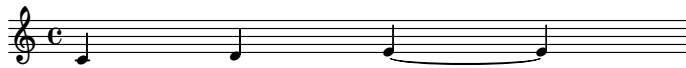
Returns string.

18.1.1.4 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
 Is 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
 Is true when logical tie head is a note or chord.
 Returns boolean.

LogicalTie.is_trivial

Is 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.tail

Reference to element -1 in logical tie.

Returns component.

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*)

Is true when *expr* is in selection. Otherwise false.

Returns boolean.

(Selection).**__eq__**(*expr*)

Is 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).**__hash__**()

Hashes selection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(Selection).**__illustrate__**()

Attempts to illustrate selection.

Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to 'RhythmicStaff'. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.

The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.

Returns LilyPond file.

(Selection).**__len__**()

Number of components in selection.

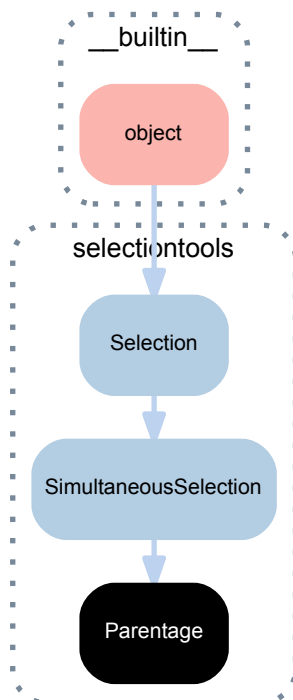
Returns nonnegative integer.

(Selection) .**__ne__**(*expr*)
Is 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.

18.1.5 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('c4')
<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`

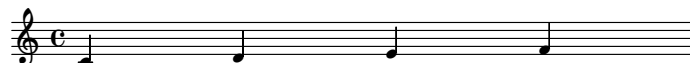
Is true when component has no parent. Otherwise false.

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.items():
...     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)
```



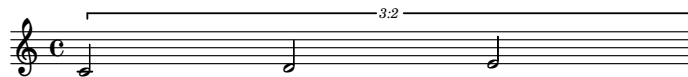
```
>>> leaves = score.select_leaves(allow_discontiguous_leaves=True)
>>> for leaf in leaves:
...     parentage = inspect_(leaf).get_parentage()
...     leaf, parentage.score_index
...
(Note("c'2"), (0, 0, 0))
(Note("b'2"), (0, 0, 1))
(Note("a'2"), (0, 0, 2))
(Note("c'2"), (1, 0))
(Note("d'2"), (1, 1))
```

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*)
 Is true when *expr* is in selection. Otherwise false.

Returns boolean.

(Selection).**__eq__** (*expr*)
 Is 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).**__hash__** ()
 Hashes selection.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 Returns integer.

(Selection).**__illustrate__** ()
 Attempts to illustrate selection.
 Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to 'RhythmicStaff'. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.
 The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.
 Returns LilyPond file.

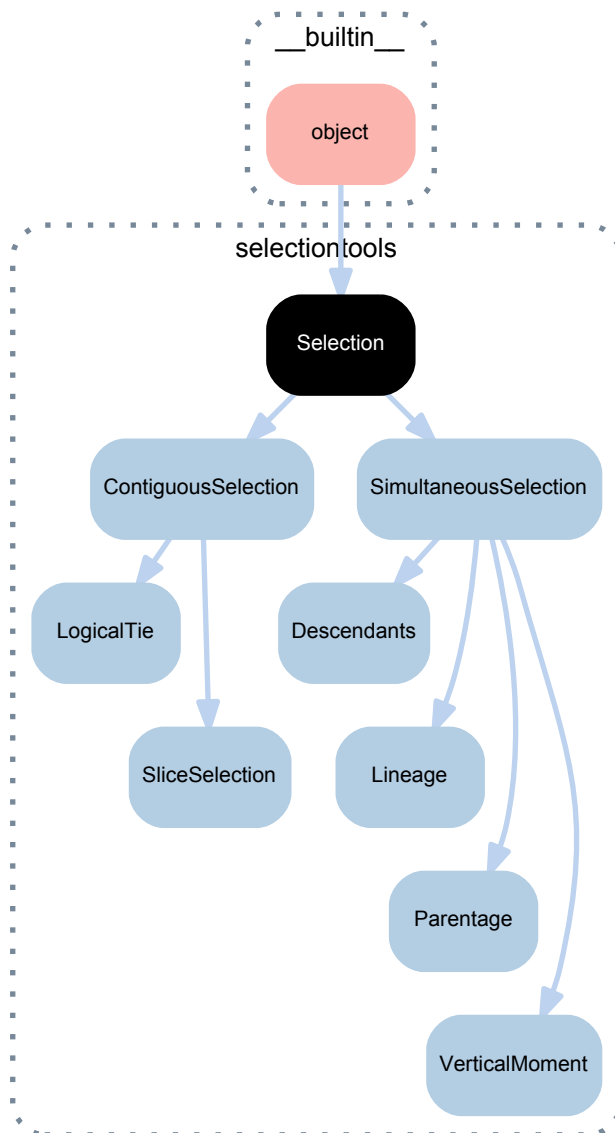
(Selection).**__len__** ()
 Number of components in selection.
 Returns nonnegative integer.

(Selection).**__ne__** (*expr*)
 Is 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.

18.1.6 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*)
 Is true when *expr* is in selection. Otherwise false.
 Returns boolean.

`Selection.__eq__` (*expr*)
 Is 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.__hash__` ()
 Hashes selection.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 Returns integer.

`Selection.__illustrate__` ()
 Attempts to illustrate selection.
 Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to *'RhythmicStaff'*. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.
 The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.
 Returns LilyPond file.

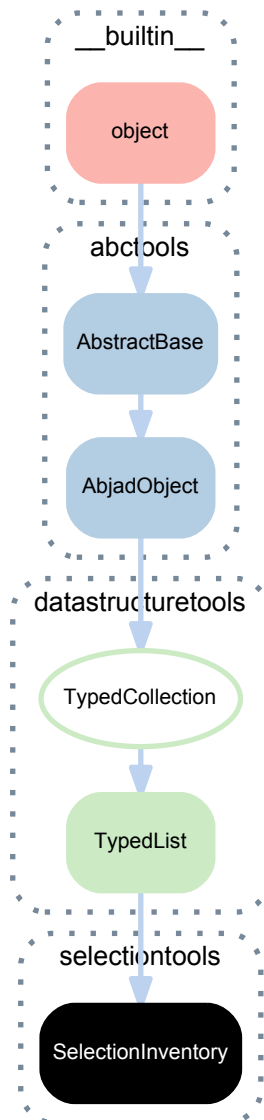
`Selection.__len__` ()
 Number of components in selection.
 Returns nonnegative integer.

`Selection.__ne__` (*expr*)
 Is 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.

18.1.7 selectiontools.SelectionInventory



class `selectiontools.SelectionInventory` (*items=None*, *item_class=None*,
keep_sorted=None)
 An inventory of component selections.

```
>>> inventory = selectiontools.SelectionInventory()
```

Bases

- `datastructuretools.TypedList`
- `datastructuretools.TypedCollection`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`

- `__builtin__.object`

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

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

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

Returns count.

`(TypedList).extend(items)`
Changes *items* 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(item)`
Changes *item* to item and returns index.

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

```
>>> pitch_collection.index("as'")
1
```

Returns index.

(TypedList) **.insert** (*i*, *item*)
Changes *item* 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** (*item*)
Changes *item* 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__** (*item*)
Is true when typed collection container *item*. Otherwise false.
Returns boolean.

(TypedList) **.__delitem__** (*i*)
Aliases list.__delitem__().
Returns none.

(TypedCollection) **.__eq__** (*expr*)
Is 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__()`.

Returns item.

(`TypedCollection`) . **__hash__** ()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(`TypedList`) . **__iadd__** (*expr*)

Changes items 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 (
            name='ppp',
        ),
        indicatortools.Dynamic (
            name='p',
        ),
        indicatortools.Dynamic (
            name='mp',
        ),
        indicatortools.Dynamic (
            name='mf',
        ),
        indicatortools.Dynamic (
            name='fff',
        ),
    ],
    item_class=indicatortools.Dynamic,
)
```

Returns collection.

(`TypedCollection`) . **__iter__** ()

Iterates typed collection.

Returns generator.

(`TypedCollection`) . **__len__** ()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__()`.

Returns generator.

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

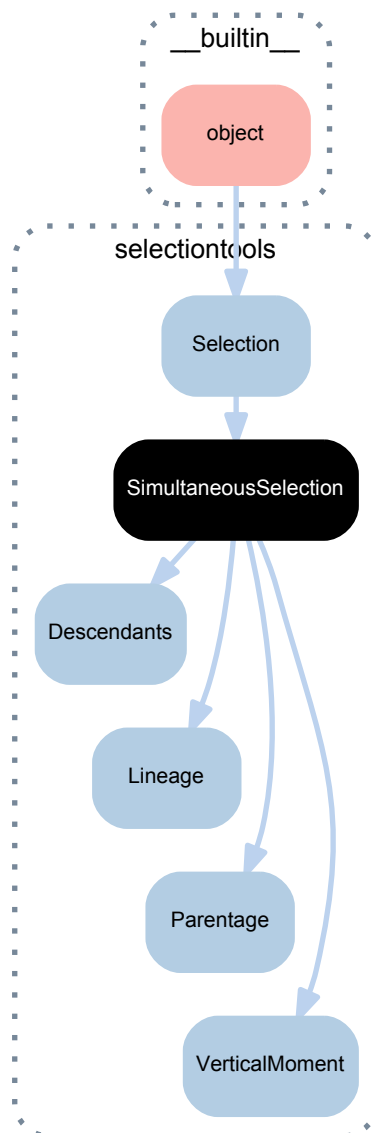
Changes items in *expr* to items and sets.

```
>>> 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,
)
```

Returns none.

18.1.8 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)`
Is true when *expr* is in selection. Otherwise false.

Returns boolean.

`(Selection).__eq__(expr)`
Is 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).__hash__()`
Hashes selection.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
Returns integer.

`(Selection).__illustrate__()`
Attempts to illustrate selection.
Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to 'RhythmicStaff'. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.

The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.

Returns LilyPond file.

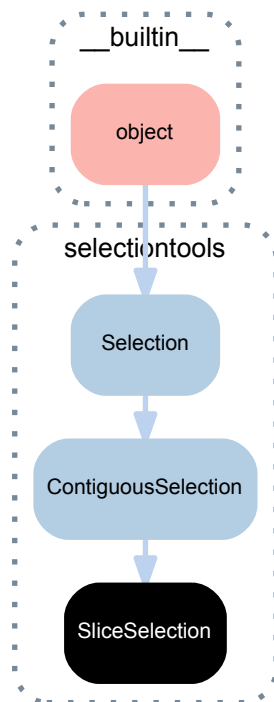
`(Selection).__len__()`
Number of components in selection.
Returns nonnegative integer.

`(Selection).__ne__(expr)`
Is true when selection does not equal *expr*. Otherwise false.
Returns boolean.

`(Selection).__radd__(expr)`
Concatenates selection to *expr*.
Returns newly created selection.

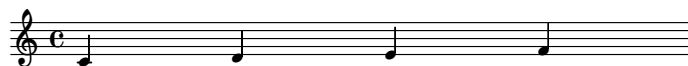
`(Selection).__repr__()`
 Gets interpreter representation of selection.
 Returns string.

18.1.9 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*)

Is true when *expr* is in selection. Otherwise false.

Returns boolean.

(Selection).**__eq__** (*expr*)

Is 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) .**__hash__** ()

Hashes selection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(Selection) .**__illustrate__** ()

Attempts to illustrate selection.

Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to 'RhythmicStaff'. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.

The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.

Returns LilyPond file.

(Selection) .**__len__** ()

Number of components in selection.

Returns nonnegative integer.

(Selection) .**__ne__** (*expr*)

Is true when selection does not equal *expr*. Otherwise false.

Returns boolean.

(ContiguousSelection) .**__radd__** (*expr*)

Adds selection to *expr*.

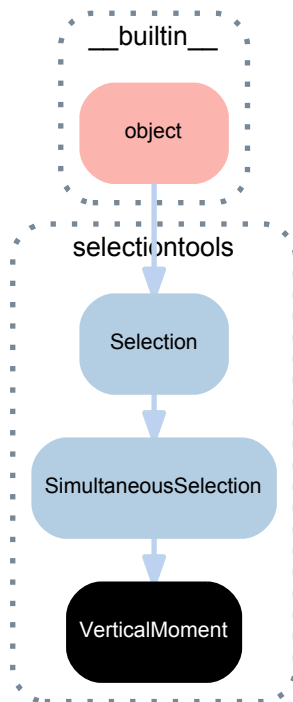
Returns new selection.

(Selection) .**__repr__** ()

Gets interpreter representation of selection.

Returns string.

18.1.10 selectiontools.VerticalMoment



class selectiontools.**VerticalMoment** (*music=None, offset=None*)
 Everything happening at a single moment in musical time:

```
>>> score = Score([])
>>> staff_group = StaffGroup()
>>> staff_group.context_name = 'PianoStaff'
>>> staff_group.append(Staff("c'4 e'4 d'4 f'4"))
>>> staff_group.append(Staff(r"""\clef "bass" g2 f2"""))
>>> score.append(staff_group)
```

```
>>> 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.music`

Gets music of vertical moment.

Returns component or selection.

`VerticalMoment.next_vertical_moment`

Reference to next vertical moment forward in time.

`VerticalMoment.notes`

Tuple of zero or more notes at vertical moment.

`VerticalMoment.notes_and_chords`

Tuple of zero or more notes and chords 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*)
 Is true when *expr* is in selection. Otherwise false.
 Returns boolean.

VerticalMoment.**__eq__** (*expr*)
 Is 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.

(Selection).**__illustrate__** ()
 Attempts to illustrate selection.
 Evaluates the storage format of the selection (to sever any references to the source score from which the selection was taken). Then tries to wrap the result in a staff; in the case that notes of only C4 are found then sets the staff context name to `'RhythmicStaff'`. If this works then the staff is wrapped in a LilyPond file and the file is returned. If this doesn't work then the method raises an exception.
 The idea is that the illustration should work for simple selections of that represent an essentially contiguous snippet of a single voice of music.
 Returns LilyPond file.

VerticalMoment.**__len__** ()
 Length of vertical moment.
 Defined equal to the number of components in vertical moment.
 Returns nonnegative integer.

VerticalMoment.**__ne__** (*expr*)
 Is true when *expr* does not equal this vertical moment. Otherwise false.
 Returns boolean.

(Selection) .__**radd**__(*expr*)

Concatenates selection to *expr*.

Returns newly created selection.

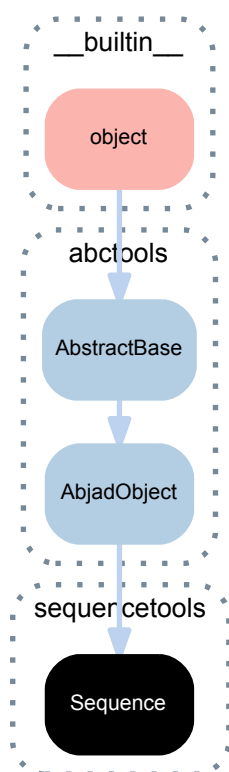
VerticalMoment .__**repr**__()

Gets interpreter representation of vertical moment.

Returns string.

19.1 Concrete classes

19.1.1 sequencetools.Sequence



```
class sequencetools.Sequence(*args)
    A sequence.
```

Bases

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

Read-only properties

```
Sequence.degree_of_rotational_symmetry
    Gets degree of rotational symmetry.
```

```
>>> Sequence(1, 1, 1, 1, 1, 1).degree_of_rotational_symmetry
6
```

```
>>> Sequence(1, 2, 1, 2, 1, 2).degree_of_rotational_symmetry
3
```

```
>>> Sequence(1, 2, 3, 1, 2, 3).degree_of_rotational_symmetry
2
```

```
>>> Sequence(1, 2, 3, 4, 5, 6).degree_of_rotational_symmetry
1
```

```
>>> Sequence().degree_of_rotational_symmetry
1
```

Returns positive integer.

Sequence.period_of_rotation

Gets period of rotation.

```
>>> Sequence(1, 2, 3, 4, 5, 6).period_of_rotation
6
```

```
>>> Sequence(1, 2, 3, 1, 2, 3).period_of_rotation
3
```

```
>>> Sequence(1, 2, 1, 2, 1, 2).period_of_rotation
2
```

```
>>> Sequence(1, 1, 1, 1, 1, 1).period_of_rotation
1
```

```
>>> Sequence().period_of_rotation
0
```

Defined equal to length of sequence divided by degree of rotational symmetry of sequence.

Returns positive integer.

Methods

Sequence.is_decreasing (*strict=True*)

Is true when sequence decreases.

With *strict=True*:

```
>>> Sequence(5, 4, 3, 2, 1, 0).is_decreasing(strict=True)
True
```

```
>>> Sequence(3, 3, 3, 2, 1, 0).is_decreasing(strict=True)
False
```

```
>>> Sequence(3, 3, 3, 3, 3, 3).is_decreasing(strict=True)
False
```

```
>>> Sequence().is_decreasing(strict=True)
True
```

With *strict=False*:

```
>>> Sequence(5, 4, 3, 2, 1, 0).is_decreasing(strict=False)
True
```

```
>>> Sequence(3, 3, 3, 2, 1, 0).is_decreasing(strict=False)
True
```

```
>>> Sequence(3, 3, 3, 3, 3, 3).is_decreasing(strict=False)
True
```

```
>>> Sequence().is_decreasing(strict=False)
True
```

Returns boolean.

Sequence.**is_increasing** (*strict=True*)
Is true when sequence increases.

With strict=True:

```
>>> Sequence(0, 1, 2, 3, 4, 5).is_increasing(strict=True)
True
```

```
>>> Sequence(0, 1, 2, 3, 3, 3).is_increasing(strict=True)
False
```

```
>>> Sequence(3, 3, 3, 3, 3, 3).is_increasing(strict=True)
False
```

```
>>> Sequence().is_increasing(strict=True)
True
```

With strict=False:

```
>>> Sequence(0, 1, 2, 3, 4, 5).is_increasing(strict=False)
True
```

```
>>> Sequence(0, 1, 2, 3, 3, 3).is_increasing(strict=False)
True
```

```
>>> Sequence(3, 3, 3, 3, 3, 3).is_increasing(strict=False)
True
```

```
>>> Sequence().is_increasing(strict=False)
True
```

Returns boolean.

Sequence.**is_permutation** (*length=None*)
Is true when sequence is a permutation.

Is true when sequence is a permutation:

```
>>> Sequence(4, 5, 0, 3, 2, 1).is_permutation()
True
```

Is false when sequence is not a permutation:

```
>>> Sequence(1, 1, 5, 3, 2, 1).is_permutation()
False
```

Returns boolean.

Sequence.**is_repetition_free** ()
Is true when sequence is repetition-free.

Is true when sequence contains no repetitions:

```
>>> Sequence(0, 1, 2, 6, 7, 8).is_repetition_free()
True
```

Is true when sequence is empty:

```
>>> Sequence().is_repetition_free()
True
```

Is false when sequence contains repetitions:

```
>>> Sequence(0, 1, 2, 2, 7, 8).is_repetition_free()
False
```

Returns boolean.

Sequence.**is_restricted_growth_function**()

Is true when sequence is a restricted growth function.

Is true when sequence is a restricted growth function:

```
>>> Sequence(1, 1, 1, 1).is_restricted_growth_function()
True
```

```
>>> Sequence(1, 1, 1, 2).is_restricted_growth_function()
True
```

```
>>> Sequence(1, 1, 2, 1).is_restricted_growth_function()
True
```

```
>>> Sequence(1, 1, 2, 2).is_restricted_growth_function()
True
```

Is false when sequence is not a restricted growth function:

```
>>> Sequence(1, 1, 1, 3).is_restricted_growth_function()
False
```

```
>>> Sequence(17).is_restricted_growth_function()
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.

Sequence.**reverse**()

Reverses this sequence.

```
>>> sequencetools.Sequence(1, 2, 3, 4, 5).reverse()
Sequence(5, 4, 3, 2, 1)
```

Emits new sequence.

Sequence.**rotate**(*n*)

Rotates this sequence.

Rotates *sequence* to the right:

```
>>> sequencetools.Sequence(*list(range(10))).rotate(4)
Sequence(6, 7, 8, 9, 0, 1, 2, 3, 4, 5)
```

Rotates *sequence* to the left:

```
>>> sequencetools.Sequence(*list(range(10))).rotate(-3)
Sequence(3, 4, 5, 6, 7, 8, 9, 0, 1, 2)
```

Rotates *sequence* neither to the right nor the left:

```
>>> sequencetools.Sequence(*list(range(10))).rotate(0)
Sequence(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Emits new sequence.

Special methods

Sequence.**__add__**(*expr*)

Adds *expr* to sequence.

Adds sequence to sequence:

```
>>> Sequence(1, 2, 3) + Sequence(4, 5, 6)
Sequence(1, 2, 3, 4, 5, 6)
```

Adds tuple to sequence:

```
>>> Sequence(1, 2, 3) + (4, 5, 6)
Sequence(1, 2, 3, 4, 5, 6)
```

Adds list to sequence:

```
>>> Sequence(1, 2, 3) + [4, 5, 6]
Sequence(1, 2, 3, 4, 5, 6)
```

Returns new sequence.

`Sequence.__eq__(expr)`

Is true when *expr* is a sequence with elements equal to those of this sequence. Otherwise false.

```
>>> Sequence(1, 2, 3, 4, 5, 6) == Sequence(1, 2, 3, 4, 5, 6)
True
```

```
>>> Sequence(1, 2, 3, 4, 5, 6) == (1, 2, 3, 4, 5, 6)
False
```

`Sequence.__format__(format_specification='')`

Formats sequence.

Prints format:

```
>>> print(format(Sequence(1, 2, 3, 4, 5, 6)))
sequencetools.Sequence(1, 2, 3, 4, 5, 6)
```

Returns string.

`Sequence.__getitem__(i)`

Gets item *i* from sequence.

Gets last item in sequence:

```
>>> Sequence(1, 2, 3, 4, 5, 6)[-1]
6
```

Returns item.

`Sequence.__getslice__(start, stop)`

Gets slice from *start* to *stop*.

Gets last three items in sequence:

```
>>> Sequence(1, 2, 3, 4, 5, 6)[-3:]
Sequence(4, 5, 6)
```

Returns new sequence.

`Sequence.__hash__()`

Hashes sequence.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`Sequence.__len__()`

Gets length of sequence.

Gets length of six-item sequence:

```
>>> len(Sequence(1, 2, 3, 4, 5, 6))
6
```

Returns nonnegative integer.

Sequence.__ne__(*expr*)

Is true when sequence is not equal to *expr*. Otherwise false.

```
>>> Sequence(1, 2, 3, 4, 5, 6) != (1, 2, 3, 4, 5, 6)
True
```

```
>>> Sequence(1, 2, 3, 4, 5, 6) != Sequence(1, 2, 3, 4, 5, 6)
False
```

Sequence.__radd__(*expr*)

Adds sequence to *expr*.

Adds sequence to sequence:

```
>>> Sequence(1, 2, 3) + Sequence(4, 5, 6)
Sequence(1, 2, 3, 4, 5, 6)
```

Adds sequence to tuple:

```
>>> (1, 2, 3) + Sequence(4, 5, 6)
Sequence(1, 2, 3, 4, 5, 6)
```

Adds sequence to list:

```
>>> [1, 2, 3] + Sequence(4, 5, 6)
Sequence(1, 2, 3, 4, 5, 6)
```

Returns new sequence.

Sequence.__repr__()

Gets interpreter representation of sequence.

```
>>> Sequence(1, 2, 3, 4, 5, 6)
Sequence(1, 2, 3, 4, 5, 6)
```

Returns string.

19.2 Functions

19.2.1 sequencetools.flatten_sequence

sequencetools.flatten_sequence(*sequence*, *classes=None*, *depth=-1*, *indices=None*)

Flattens *sequence*.

Flattens sequence completely:

```
>>> sequence = [1, [2, 3, [4]], 5, [6, 7, [8]]]
>>> sequencetools.flatten_sequence(sequence)
[1, 2, 3, 4, 5, 6, 7, 8]
```

Flattens *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]]
```

Flattens *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]
```

Flattens *sequence* at *indices*:

```
>>> sequence = [0, 1, [2, 3, 4], [5, 6, 7]]
>>> sequencetools.flatten_sequence(sequence, indices=[3])
[0, 1, [2, 3, 4], 5, 6, 7]
```


Flattens *sequence* at negative *indices*:

```
>>> sequence = [0, 1, [2, 3, 4], [5, 6, 7]]
>>> sequencetools.flatten_sequence(sequence, indices=[-1])
[0, 1, [2, 3, 4], 5, 6, 7]
```

Leaves *sequence* unchanged.

Returns new object of *sequence* type.

19.2.2 sequencetools.increase_elements

`sequencetools.increase_elements(sequence, addenda, indices=None)`

Increases *sequence* cyclically by *addenda*.

Increases elements cyclically by 10 and -10 in alternation:

```
>>> sequencetools.increase_elements(range(10), [10, -10])
[10, -9, 12, -7, 14, -5, 16, -3, 18, -1]
```

Increases elements by 0.5 at indices 0, 4 and 8 and at one element following each:

```
>>> sequence = [1, 1, 2, 3, 5, 5, 1, 2, 5, 5, 6]
>>> sequencetools.increase_elements(
...     sequence, [0.5, 0.5], indices=[0, 4, 8])
[1.5, 1.5, 2, 3, 5.5, 5.5, 1, 2, 5.5, 5.5, 6]
```

Returns list.

19.2.3 sequencetools.interlace_sequences

`sequencetools.interlace_sequences(*sequences)`

Interlaces *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.

19.2.4 sequencetools.iterate_sequence_boustrophedon

`sequencetools.iterate_sequence_boustrophedon(sequence, duplicates=False)`

Iterates *sequence* boustrophedon.

Iterates *sequence* first forward and then backward. Duplicates neither first nor last elements:

```
>>> sequence = [1, 2, 3, 4, 5]
>>> generator = sequencetools.iterate_sequence_boustrophedon(
...     sequence, duplicates=False)
>>> list(generator)
[1, 2, 3, 4, 5, 4, 3, 2]
```

Iterates *sequence* first forward and then backward. Duplicates both first and last elements:

```
>>> sequence = [1, 2, 3, 4, 5]
>>> generator = sequencetools.iterate_sequence_boustrophedon(
...     sequence, duplicates=True)
>>> list(generator)
[1, 2, 3, 4, 5, 5, 4, 3, 2, 1]
```

Returns generator.

19.2.5 sequencetools.iterate_sequence_nwise

`sequencetools.iterate_sequence_nwise(sequence, n=2, cyclic=False, wrapped=False)`

Iterates elements in *sequence* *n* at a time.

Iterates sequence by pairs:

```
>>> for x in sequencetools.iterate_sequence_nwise(range(10)):
...     x
...
(0, 1)
(1, 2)
(2, 3)
(3, 4)
(4, 5)
(5, 6)
(6, 7)
(7, 8)
(8, 9)
```

Iterates sequence by triples:

```
>>> for x in sequencetools.iterate_sequence_nwise(range(10), n=3):
...     x
...
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(4, 5, 6)
(5, 6, 7)
(6, 7, 8)
(7, 8, 9)
```

Iterates sequence by pairs. Wraps around at end:

```
>>> for x in sequencetools.iterate_sequence_nwise(
...     range(10), n=2, wrapped=True):
...     x
...
(0, 1)
(1, 2)
(2, 3)
(3, 4)
(4, 5)
(5, 6)
(6, 7)
(7, 8)
(8, 9)
(9, 0)
```

Iterates sequence by triples. Wraps around at end:

```
>>> for x in sequencetools.iterate_sequence_nwise(
...     range(10), n=3, wrapped=True):
...     x
...
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(4, 5, 6)
(5, 6, 7)
(6, 7, 8)
(7, 8, 9)
(8, 9, 0)
(9, 0, 1)
```

Iterates sequence by pairs. Cycles indefinitely:

```
>>> pairs = sequencetools.iterate_sequence_nwise(
...     range(10), n=2, cyclic=True)
>>> for _ in range(15):
...     next(pairs)
(0, 1)
(1, 2)
(2, 3)
(3, 4)
(4, 5)
(5, 6)
(6, 7)
(7, 8)
(8, 9)
(9, 0)
(0, 1)
(1, 2)
(2, 3)
(3, 4)
(4, 5)
```

Returns infinite generator.

Iterates sequence by triples. Cycles indefinitely:

```
>>> triples = sequencetools.iterate_sequence_nwise(
...     range(10), n=3, cyclic=True)
...
>>> for _ in range(15):
...     next(triples)
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(4, 5, 6)
(5, 6, 7)
(6, 7, 8)
(7, 8, 9)
(8, 9, 0)
(9, 0, 1)
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(4, 5, 6)
```

Returns infinite generator.

Ignores wrapped when `cyclic=True`.

Returns generator.

19.2.6 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.

19.2.7 sequencetools.join_subsequences_by_sign_of_elements

`sequencetools.join_subsequences_by_sign_of_elements(sequence)`

Join subsequences in *sequence* by sign of elements.

```
>>> sequence = [[1, 2], [3, 4], [-5, -6, -7], [-8, -9, -10], [11, 12]]
>>> sequencetools.join_subsequences_by_sign_of_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_elements(sequence)
[[1, 2], [], [3, 4, 5, 6, 7]]
```

Returns new list.

19.2.8 sequencetools.negate_elements

`sequencetools.negate_elements(sequence, absolute=False, indices=None, period=None)`

Negates *sequence* elements.

Negates all elements:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
>>> sequencetools.negate_elements(sequence)
[-1, -2, -3, -4, -5, 6, 7, 8, 9, 10]
```

Negates elements at indices 0, 1 and 2:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
>>> sequencetools.negate_elements(sequence, indices=[0, 1, 2])
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Negates elements at indices congruent to 0, 1 or 2 mod 5:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
>>> sequencetools.negate_elements(
...     sequence,
...     indices=[0, 1, 2],
...     period=5,
... )
[-1, -2, -3, 4, 5, 6, 7, 8, -9, -10]
```

Negates the absolute value of all elements:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
>>> sequencetools.negate_elements(sequence, absolute=True)
[-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
```

Negates the absolute value elements at indices 0, 1 and 2:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
>>> sequencetools.negate_elements(
...     sequence,
...     absolute=True,
...     indices=[0, 1, 2],
... )
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Negates the absolute value elements at indices congruent to 0, 1 or 2 mod 5:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
>>> sequencetools.negate_elements(
...     sequence,
...     absolute=True,
...     indices=[0, 1, 2],
...     period=5,
... )
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Returns newly constructed list.

19.2.9 sequencetools.overwrite_elements

`sequencetools.overwrite_elements(sequence, pairs)`
Overwrites *sequence* elements at indices according to *pairs*.

```
>>> sequencetools.overwrite_elements(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.

19.2.10 sequencetools.partition_sequence_by_counts

`sequencetools.partition_sequence_by_counts(sequence, counts, cyclic=False, overhang=False, copy_elements=False)`
Partitions sequence by counts.

Example 1a. Partition sequence once by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     list(range(10)),
...     [3],
...     cyclic=False,
...     overhang=False,
...     )
[[0, 1, 2]]
```

Example 1b. Partition sequence once by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     list(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(
...     list(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(
...     list(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(
...     list(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(
...     list(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(
...     list(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(
...     list(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.

19.2.11 `sequencetools.partition_sequence_by_ratio_of_lengths`

`sequencetools.partition_sequence_by_ratio_of_lengths` (*sequence*, *lengths*)

Partitions *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)]
```

Uses rounding magic to avoid fractional part lengths.

Returns list of *sequence* objects.

19.2.12 `sequencetools.partition_sequence_by_ratio_of_weights`

`sequencetools.partition_sequence_by_ratio_of_weights` (*sequence*, *weights*)

Partitions *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, 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, 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.

19.2.13 sequencetools.partition_sequence_by_restricted_growth_function

`sequencetools.partition_sequence_by_restricted_growth_function` (*sequence*,
restricted_growth_function)

Partitions *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]]
```

Raises value error when *sequence* length does not equal *restricted_growth_function* length.

Returns list of lists.

19.2.14 sequencetools.partition_sequence_by_sign_of_elements

`sequencetools.partition_sequence_by_sign_of_elements` (*sequence*, *sign*=(-1, 0, 1))

Partitions *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, groups negative elements.

When 0 in sign, groups 0 elements.

When 1 in sign, groups positive elements.

Returns list of tuples of *sequence* element references.

19.2.15 sequencetools.partition_sequence_by_value_of_elements

`sequencetools.partition_sequence_by_value_of_elements(sequence)`

Groups *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.

19.2.16 sequencetools.partition_sequence_by_weights

`sequencetools.partition_sequence_by_weights(sequence, weights, cyclic=False, overhang=False, low_part_weights=Exact)`

Partitions *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(
...     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(
...     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(
...     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(
...     sequence,
...     [12],
...     cyclic=True,
...     overhang=True,
... )
[[3, 3, 3, 3], [4, 4, 4], [4, 5]]
```

```
>>> 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(
...     sequence,
...     [10, 4],
...     cyclic=False,
...     overhang=False,
...     allow_part_weights=Less,
... )
[[3, 3, 3], [3]]
```

Example 2. Partition sequence once by weights at most with overhang:

```
>>> sequencetools.partition_sequence_by_weights(
...     sequence,
...     [10, 4],
...     cyclic=False,
...     overhang=True,
...     allow_part_weights=Less,
... )
[[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(
...     sequence,
...     [10, 5],
...     cyclic=True,
...     overhang=False,
...     allow_part_weights=Less,
... )
[[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(
...     sequence,
...     [10, 5],
...     cyclic=True,
...     overhang=True,
...     allow_part_weights=Less,
... )
[[3, 3, 3], [3], [4, 4], [4], [4, 5], [5]]
```

```
>>> 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(
...     sequence,
...     [10, 4],
...     cyclic=False,
...     overhang=False,
...     allow_part_weights=More,
... )
[[3, 3, 3, 3], [4]]
```

Example 2. Partition sequence once by weights at least with overhang:

```
>>> sequencetools.partition_sequence_by_weights(
...     sequence,
...     [10, 4],
...     cyclic=False,
...     overhang=True,
...     allow_part_weights=More,
... )
[[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(
...     sequence,
...     [10, 4],
...     cyclic=True,
...     overhang=False,
...     allow_part_weights=More,
... )
[[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(
...     sequence,
...     [10, 4],
...     cyclic=True,
...     overhang=True,
...     allow_part_weights=More,
... )
[[3, 3, 3, 3], [4], [4, 4, 4], [5], [5]]
```

Returns list sequence objects.

19.2.17 `sequencetools.permute_sequence`

`sequencetools.permute_sequence(sequence, permutation)`

Permutes *sequence*.

```
>>> sequencetools.permute_sequence([10, 11, 12, 13, 14, 15], [5, 4, 0, 1, 2, 3])
[15, 14, 10, 11, 12, 13]
```

Returns new object of *sequence* type.

19.2.18 `sequencetools.remove_elements`

`sequencetools.remove_elements(sequence, indices=None, period=None)`

Removes *sequence* elements at *indices*.

Removes all elements:

```
>>> sequencetools.remove_elements(range(15))
[]
```

Removes elements and indices 2 and 3:

```
>>> sequencetools.remove_elements(
...     range(15),
...     indices=[2, 3],
... )
[0, 1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Removes elements and indices -2 and -3:

```
>>> sequencetools.remove_elements(
...     range(15),
...     indices=[-2, -3],
```

```
...    )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14]
```

Removes elements and indices 2 and 3 (mod 4):

```
>>> sequencetools.remove_elements(
...     range(15),
...     indices=[2, 3],
...     period=4,
...     )
[0, 1, 4, 5, 8, 9, 12, 13]
```

Removes elements and indices -2 and -3 (mod 4):

```
>>> sequencetools.remove_elements(
...     range(15),
...     indices=[-2, -3],
...     period=4,
...     )
[2, 3, 6, 7, 10, 11, 14]
```

Removes no elements:

```
>>> sequencetools.remove_elements(
...     range(15),
...     indices=[],
...     )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Removes no elements:

```
>>> sequencetools.remove_elements(
...     range(15),
...     indices=[97, 98, 99],
...     )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Removes no elements:

```
>>> sequencetools.remove_elements(
...     range(15),
...     indices=[-97, -98, -99],
...     )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Returns elements in the order they appear in *sequence*.

Returns list.

19.2.19 sequencetools.remove_repeated_elements

`sequencetools.remove_repeated_elements(sequence)`

Removes repeated elements from *sequence*.

```
>>> sequence = (17, 18, 18, 18, 19, 20, 21, 22, 22, 24, 24)
>>> sequencetools.remove_repeated_elements(sequence)
(17, 18, 19, 20, 21, 22, 24)
```

```
>>> sequence = [31, 31, 35, 35, 31, 31, 31, 31, 35]
>>> sequencetools.remove_repeated_elements(sequence)
[31, 35, 31, 35]
```

Returns new object of *sequence* type.

19.2.20 sequencetools.remove_subsequence_of_weight_at_index

`sequencetools.remove_subsequence_of_weight_at_index(sequence, weight, index)`
Removes 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.

19.2.21 sequencetools.repeat_elements

`sequencetools.repeat_elements(sequence, indices=None, period=None, total=1)`
Repeats *sequence* elements.

Repeats elements at indices 1 and 2 a total of three times each:

```
>>> sequencetools.repeat_elements(
...     list(range(10)),
...     indices=[1, 2],
...     total=3,
... )
[0, [1, 1, 1], [2, 2, 2], 3, 4, 5, 6, 7, 8, 9]
```

Repeats elements at indices -1 and -2 a total of three times each:

```
>>> sequencetools.repeat_elements(
...     list(range(10)),
...     indices=[-1, -2],
...     total=3,
... )
[0, 1, 2, 3, 4, 5, 6, 7, [8, 8, 8], [9, 9, 9]]
```

Repeats elements at indices congruent to 1 and 2 (mod 5) a total of three times each:

```
>>> sequencetools.repeat_elements(
...     list(range(10)),
...     indices=[1, 2],
...     total=3,
...     period=5,
... )
[0, [1, 1, 1], [2, 2, 2], 3, 4, 5, [6, 6, 6], [7, 7, 7], 8, 9]
```

Repeats elements at indices congruent to -1 and -2 (mod 5) a total of three times each:

```
>>> sequencetools.repeat_elements(
...     list(range(10)),
...     indices=[-1, -2],
...     total=3,
...     period=5,
... )
[0, 1, 2, [3, 3, 3], [4, 4, 4], 5, 6, 7, [8, 8, 8], [9, 9, 9]]
```

Repeats all elements a total of two times each:

```
>>> sequencetools.repeat_elements(
...     list(range(10)),
...     total=2,
... )
[[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [6, 6], [7, 7], [8, 8], [9, 9]]
```

Repeats all elements a total of one time each:

```
>>> sequencetools.repeat_elements(
...     list(range(10)),
...     total=1,
... )
```

```
...    )
[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9]]
```

Repeats all elements a total of zero times each:

```
>>> sequencetools.repeat_elements(
...     list(range(10)),
...     total=0,
...     )
[[], [], [], [], [], [], [], [], [], []]
```

Repeats no elements:

```
>>> sequencetools.repeat_elements(
...     list(range(10)),
...     indices=[],
...     )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Returns new object of *sequence* type.

19.2.22 sequencetools.repeat_sequence

`sequencetools.repeat_sequence(sequence, n)`

Repeats *sequence* *n* times.

```
>>> sequencetools.repeat_sequence((1, 2, 3, 4, 5), 3)
(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

Repeats *sequence* 0 times:

```
>>> sequencetools.repeat_sequence((1, 2, 3, 4, 5), 0)
()
```

Returns newly constructed *sequence* object of copied *sequence* elements.

19.2.23 sequencetools.repeat_sequence_to_length

`sequencetools.repeat_sequence_to_length(sequence, length, start=0)`

Repeats *sequence* to nonnegative integer *length*.

```
>>> sequencetools.repeat_sequence_to_length(list(range(5)), 11)
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0]
```

Repeats *sequence* to nonnegative integer *length* from *start*:

```
>>> sequencetools.repeat_sequence_to_length(
...     list(range(5)), 11, start=2)
[2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2]
```

Returns newly constructed *sequence* object.

19.2.24 sequencetools.repeat_sequence_to_weight

`sequencetools.repeat_sequence_to_weight(sequence, weight, allow_total=Exact)`

Repeats *sequence* to *weight*.

Repeats sequence to weight of 23 exactly:

```
>>> sequencetools.repeat_sequence_to_weight((5, -5, -5), 23)
(5, -5, -5, 5, -3)
```

Truncates last element when necessary.

Repeats sequence to weight of 23 more:

```
>>> sequencetools.repeat_sequence_to_weight(  
...     (5, -5, -5),  
...     23,  
...     allow_total=More,  
... )  
(5, -5, -5, 5, -5)
```

Does not truncate last element.

Repeats sequence to weight of 23 or less:

```
>>> sequencetools.repeat_sequence_to_weight(  
...     (5, -5, -5),  
...     23,  
...     allow_total=Less,  
... )  
(5, -5, -5, 5)
```

Discards last element when necessary.

Returns newly constructed *sequence* object.

19.2.25 sequencetools.replace_elements

`sequencetools.replace_elements(sequence, indices, new_material)`

Replaces *sequence* elements.

Replaces elements at indices 0, 2, 4, 6 with 'A', 'B', 'C', 'D', respectively:

```
>>> sequencetools.replace_elements(  
...     list(range(16)),  
...     ([0, 2],  
...     (['A', 'B', 'C', 'D'], None),  
... )  
['A', 1, 'B', 3, 'C', 5, 'D', 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Replaces elements at indices 0, 1, 8, 13 with 'A', 'B', 'C', 'D', respectively:

```
>>> sequencetools.replace_elements(  
...     list(range(16)),  
...     ([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]
```

Replaces every element at an even index with '*':

```
>>> sequencetools.replace_elements(  
...     list(range(16)),  
...     ([0, 2],  
...     (['*'], 1),  
... )  
['*', 1, '*', 3, '*', 5, '*', 7, '*', 9, '*', 11, '*', 13, '*', 15]
```

Replaces every element at an index congruent to 0 (mod 6) with 'A'; replaces every element at an index congruent to 2 (mod 6) with 'B':

```
>>> sequencetools.replace_elements(  
...     list(range(16)),  
...     ([0, 2],  
...     (['A', 'B'], 3),  
... )  
['A', 1, 'B', 3, 4, 5, 'A', 7, 'B', 9, 10, 11, 'A', 13, 'B', 15]
```

Raises type error when *sequence* is not a list.

Returns object of *sequence* type.

19.2.26 sequencetools.retain_elements

`sequencetools.retain_elements(sequence, indices=None, period=None)`

Retains *sequence* elements.

Retains all elements:

```
>>> sequencetools.retain_elements(range(15))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Retains elements at indices 2 and 3:

```
>>> sequencetools.retain_elements(
...     range(15),
...     indices=[2, 3],
...     )
[2, 3]
```

Retains elements at indices -2 and -3:

```
>>> sequencetools.retain_elements(
...     range(15),
...     indices=[-2, -3],
...     )
[12, 13]
```

Retains elements at indices congruent to 2 or 3 (mod 4):

```
>>> sequencetools.retain_elements(
...     range(15),
...     indices=[2, 3],
...     period=4,
...     )
[2, 3, 6, 7, 10, 11, 14]
```

Retains elements at indices congruent to -2 or -3 (mod 4):

```
>>> sequencetools.retain_elements(
...     range(15),
...     indices=[-2, -3],
...     period=4,
...     )
[0, 1, 4, 5, 8, 9, 12, 13]
```

Retains no elements:

```
>>> sequencetools.retain_elements(
...     range(15),
...     indices=[],
...     )
[]
```

Retains no elements:

```
>>> sequencetools.retain_elements(
...     range(15),
...     indices=[97, 98, 99],
...     )
[]
```

Retains no elements:

```
>>> sequencetools.retain_elements(
...     range(15),
...     indices=[-97, -98, -99],
...     )
[]
```

Returns elements in the order they appear in *sequence*.

Returns list.

19.2.27 sequencetools.reverse_sequence

`sequencetools.reverse_sequence(sequence)`

Reverses *sequence*.

```
>>> sequencetools.reverse_sequence((1, 2, 3, 4, 5))
(5, 4, 3, 2, 1)
```

Returns new *sequence* object.

19.2.28 sequencetools.rotate_sequence

`sequencetools.rotate_sequence(sequence, n)`

Rotates *sequence*.

Rotates *sequence* to the right:

```
>>> sequencetools.rotate_sequence(list(range(10)), 4)
[6, 7, 8, 9, 0, 1, 2, 3, 4, 5]
```

Rotates *sequence* to the left:

```
>>> sequencetools.rotate_sequence(list(range(10)), -3)
[3, 4, 5, 6, 7, 8, 9, 0, 1, 2]
```

Rotates *sequence* neither to the right nor the left:

```
>>> sequencetools.rotate_sequence(list(range(10)), 0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Returns newly created *sequence* object.

19.2.29 sequencetools.splice_between_elements

`sequencetools.splice_between_elements(sequence, new_elements, overhang=(0, 0))`

Splices copies of *new_elements* between each of the elements of *sequence*.

```
>>> sequence = [0, 1, 2, 3, 4]
>>> new_elements = ['A', 'B']
```

```
>>> sequencetools.splice_between_elements(sequence, new_elements)
[0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4]
```

Splices copies of *new_elements* between each of the elements of *sequence* and after the last element of *sequence*:

```
>>> sequencetools.splice_between_elements(
...     sequence, new_elements, overhang=(0, 1))
[0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4, 'A', 'B']
```

Splices copies of *new_elements* before the first element of *sequence* and between each of the other elements of *sequence*:

```
>>> sequencetools.splice_between_elements(
...     sequence, new_elements, overhang=(1, 0))
['A', 'B', 0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4]
```

Splices 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_between_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.

19.2.30 sequencetools.split_sequence

`sequencetools.split_sequence(sequence, weights, cyclic=False, overhang=False)`

Splits sequence by weights.

Example 1. Split sequence cyclically by weights with overhang:

```
>>> sequencetools.split_sequence(
...     (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(
...     (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(
...     (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(
...     (10, -10, 10, -10),
...     (3, 15, 3),
...     cyclic=False,
...     overhang=False,
... )
[(3,), (7, -8), (-2, 1)]
```

Returns list of sequence types.

19.2.31 sequencetools.sum_consecutive_elements_by_sign

`sequencetools.sum_consecutive_elements_by_sign(sequence, sign=(-1, 0, 1))`

Sums consecutive *sequence* elements by *sign*.

```
>>> sequence = [0, 0, -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> sequencetools.sum_consecutive_elements_by_sign(sequence)
[0, -2, 5, -5, 8, -11]
```

```
>>> sequencetools.sum_consecutive_elements_by_sign(sequence, sign=[-1])
[0, 0, -2, 2, 3, -5, 1, 2, 5, -11]
```

```
>>> sequencetools.sum_consecutive_elements_by_sign(sequence, sign=[0])
[0, -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> sequencetools.sum_consecutive_elements_by_sign(sequence, sign=[1])
[0, 0, -1, -1, 5, -5, 8, -5, -6]
```

```
>>> sequencetools.sum_consecutive_elements_by_sign(sequence, sign=[-1, 0])
[0, -2, 2, 3, -5, 1, 2, 5, -11]
```

```
>>> sequencetools.sum_consecutive_elements_by_sign(sequence, sign=[-1, 1])
[0, 0, -2, 5, -5, 8, -11]
```

```
>>> sequencetools.sum_consecutive_elements_by_sign(sequence, sign=[0, 1])
[0, -1, -1, 5, -5, 8, -5, -6]
```

```
>>> sequencetools.sum_consecutive_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.

19.2.32 sequencetools.sum_elements

`sequencetools.sum_elements(sequence, pairs, period=None, overhang=True)`
Sums *sequence* elements at indices according to *pairs*.

```
>>> sequencetools.sum_elements(list(range(10)), [(0, 3)])
[3, 3, 4, 5, 6, 7, 8, 9]
```

Sums *sequence* elements cyclically at indices according to *pairs* and *period*:

```
>>> sequencetools.sum_elements(list(range(10)), [(0, 3)], period=4)
[3, 3, 15, 7, 17]
```

Sums *sequence* elements cyclically at indices according to *pairs* and *period* and do not return incomplete final sum:

```
>>> sequencetools.sum_elements(
...     list(range(10)), [(0, 3)], period=4, overhang=False)
[3, 3, 15, 7]
```

Replaces `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.

19.2.33 sequencetools.truncate_sequence

`sequencetools.truncate_sequence(sequence, sum_=None, weight=None)`
Truncates *sequence*.

Example sequence:

```
>>> sequence = [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
```

Truncates sequence to weights ranging from 1 to 10:

```
>>> for n in range(1, 11):
...     result = sequencetools.truncate_sequence(sequence, weight=n)
...     print(n, result)
...
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]
10 [-1, 2, -3, 4]
```

Truncates sequence to sums ranging from 1 to 10:

```
>>> for n in range(1, 11):
...     result = sequencetools.truncate_sequence(sequence, sum_=n)
...     print(n, result)
...
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]
10 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
```

Truncates sequence to zero weight:

```
>>> sequencetools.truncate_sequence(sequence, weight=0)
[]
```

Truncates sequence to zero sum:

```
>>> sequencetools.truncate_sequence(sequence, sum_=0)
[]
```

Ignores *sum_* when *weight* and *sum_* are both set.

Raises type error when *sequence* is not a list.

Raises value error on negative *sum_*.

Returns new object of *sequence* type.

19.2.34 sequencetools.yield_all_combinations_of_elements

`sequencetools.yield_all_combinations_of_elements` (*sequence*, *min_length=None*, *max_length=None*)

Yields all combinations of *sequence* in binary string order.

```
>>> list(sequencetools.yield_all_combinations_of_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]]
```

Yields all combinations of *sequence* greater than or equal to *min_length* in binary string order:

```
>>> list(sequencetools.yield_all_combinations_of_elements(
...     [1, 2, 3, 4], min_length=3))
[[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

Yields all combinations of *sequence* less than or equal to *max_length* in binary string order:

```
>>> list(sequencetools.yield_all_combinations_of_elements(
...     [1, 2, 3, 4], max_length=2))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [4], [1, 4], [2, 4], [3, 4]]
```

Yields 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_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.

19.2.35 sequencetools.yield_all_k_ary_sequences_of_length

`sequencetools.yield_all_k_ary_sequences_of_length(k, length)`

Yields 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.

19.2.36 sequencetools.yield_all_pairs_between_sequences

`sequencetools.yield_all_pairs_between_sequences(l, m)`

Yields 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.

19.2.37 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.

19.2.38 sequencetools.yield_all_permutations_of_sequence

`sequencetools.yield_all_permutations_of_sequence(sequence)`

Yields all permutations of *sequence*.

```
>>> 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 permutations in lex order.

Returns generator of *sequence* objects.

19.2.39 sequencetools.yield_all_permutations_of_sequence_in_orbit

`sequencetools.yield_all_permutations_of_sequence_in_orbit(sequence, permutation)`

Yields all permutations of *sequence* in orbit of *permutation*.

```
>>> 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 permutations in lex order.

Returns generator of *sequence* objects.

19.2.40 sequencetools.yield_all_restricted_growth_functions_of_length

`sequencetools.yield_all_restricted_growth_functions_of_length(length)`

Yields all restricted growth functions of *length*.

```
>>> 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 restricted growth functions in lex order.

Returns generator of tuples.

19.2.41 sequencetools.yield_all_rotations_of_sequence

`sequencetools.yield_all_rotations_of_sequence(sequence, n=1)`

Yields all n -rotations of *sequence*.

```
>>> 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]]
```

Yields rotations up to but not including identity.

Returns generator of *sequence* objects.

19.2.42 sequencetools.yield_all_set_partitions_of_sequence

`sequencetools.yield_all_set_partitions_of_sequence(sequence)`

Yields all set partitions of *sequence*.

```
>>> 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 set partitions in order of restricted growth function.

Returns generator of list of lists.

19.2.43 sequencetools.yield_all_subsequences_of_sequence

`sequencetools.yield_all_subsequences_of_sequence(sequence, min_length=0, max_length=None)`

Yields all subsequences of *sequence*.

Yields subsequences in lex order.

```
>>> list(sequencetools.yield_all_subsequences_of_sequence([0, 1, 2]))
[[], [0], [0, 1], [0, 1, 2], [1], [1, 2], [2]]
```

Yields 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]]
```

Yields 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]
```

Yields 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.

19.2.44 sequencetools.yield_all_unordered_pairs_of_sequence

`sequencetools.yield_all_unordered_pairs_of_sequence(sequence)`

Yields 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)]
```

Yields all unordered pairs of length-1 *sequence*:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([1]))
[]
```

Yields all unordered pairs of empty *sequence*:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([]))
[]
```

Yields 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.

19.2.45 sequencetools.yield_outer_product_of_sequences

`sequencetools.yield_outer_product_of_sequences(sequences)`

Yields 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.

19.2.46 sequencetools.zip_sequences

`sequencetools.zip_sequences` (*sequences*, *cyclic=False*, *truncate=True*)

Zips *sequences*.

Zips two *sequences* cyclically:

```
>>> sequences = [[1, 2, 3], ['a', 'b']]
>>> sequencetools.zip_sequences(sequences, cyclic=True)
[(1, 'a'), (2, 'b'), (3, 'a')]
```

Zips three *sequences* cyclically:

```
>>> sequences = [[10, 11, 12], [20, 21], [30, 31, 32, 33]]
>>> sequencetools.zip_sequences(sequences, cyclic=True)
[(10, 20, 30), (11, 21, 31), (12, 20, 32), (10, 21, 33)]
```

Zips sequences without truncation:

```
>>> sequences = ([1, 2, 3, 4], [11, 12, 13], [21, 22, 23])
>>> sequencetools.zip_sequences(sequences, truncate=False)
[(1, 11, 21), (2, 12, 22), (3, 13, 23), (4,)]
```

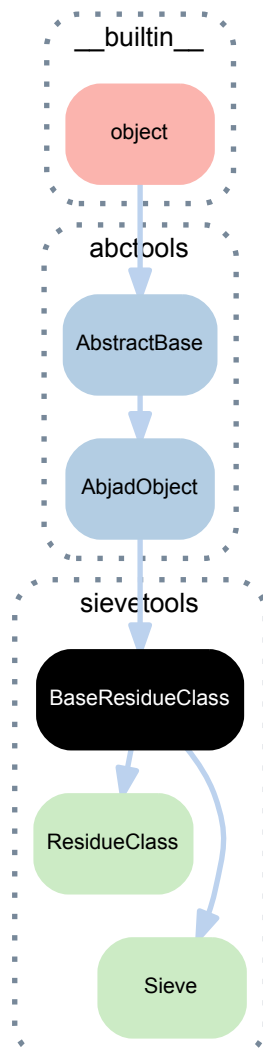
Zips sequences noncyclically and with truncation. Equivalent to built-in `zip()`:

```
>>> sequences = ([1, 2, 3, 4], [11, 12, 13], [21, 22, 23])
>>> sequencetools.zip_sequences(sequences)
[(1, 11, 21), (2, 12, 22), (3, 13, 23)]
```

Returns list.

20.1 Concrete classes

20.1.1 `sievetools.BaseResidueClass`



class `sievetools.BaseResidueClass`
Abstract base class for `ResidueClass` and `Sieve`.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(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.

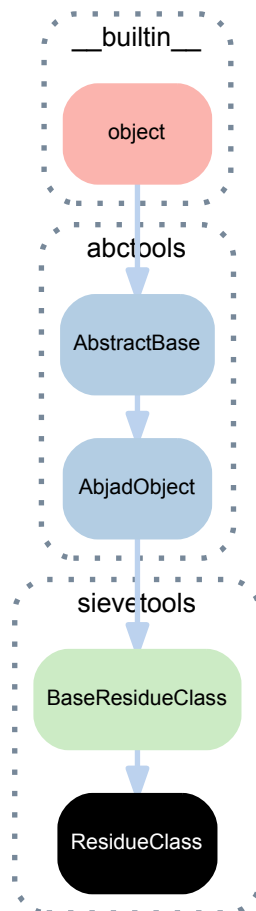
Returns string.

`BaseResidueClass.__xor__(arg)`

Logical XOR of residue class and *arg*.

Returns sieve.

20.1.2 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, 1, 0]
```

Bases

- `sievetools.BaseResidueClass`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`ResidueClass.module`
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)`
Is 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 Abjad object.

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

Returns string.

ResidueClass.**__ge__**(*other*)

$x._\text{ge}_(y) \iff x \geq y$

ResidueClass.**__gt__**(*other*)

$x._\text{gt}_(y) \iff x > y$

ResidueClass.**__hash__**()

Hashes residue class.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

ResidueClass.**__le__**(*other*)

$x._\text{le}_(y) \iff x \leq y$

ResidueClass.**__lt__**(*expr*)

Is 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*)

Is 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.

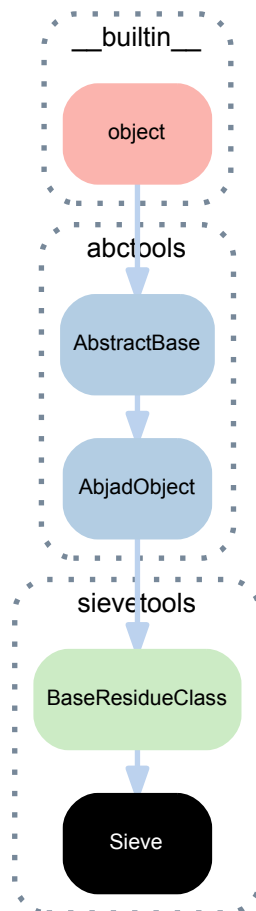
Returns string.

(BaseResidueClass).**__xor__**(*arg*)

Logical XOR of residue class and *arg*.

Returns sieve.

20.1.3 sievetools.Sieve



class `sieve tools.Sieve` (*rsc=None, logical_operator='or'*)
 A Xenakis sieve.

Bases

- `sieve tools.BaseResidueClass`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`Sieve.logical_operator`
 Residue class expression logical operator.

`Sieve.period`
 Residue class expression period.

`Sieve.rsc`
 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)`

Is 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 Abjad object.

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

Returns string.

(AbjadObject) .**__hash__**()
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(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.

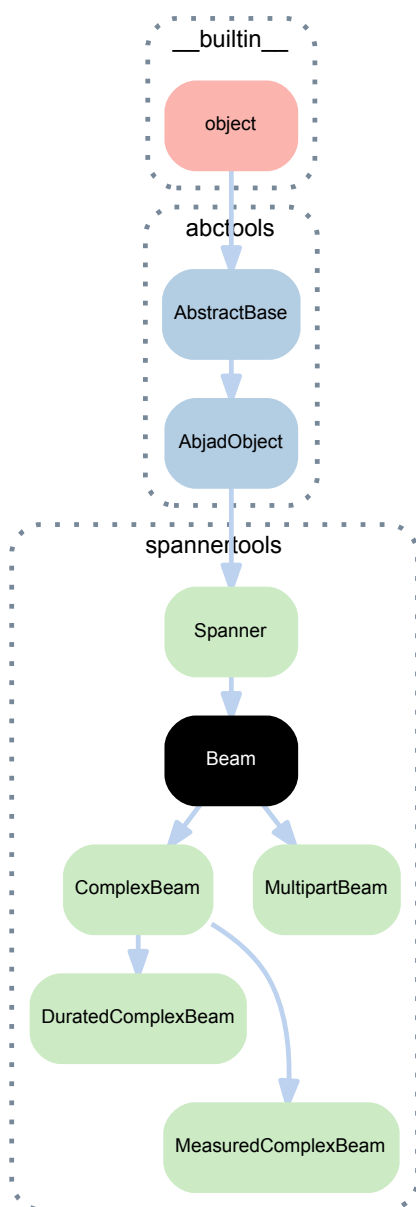
Returns string.

(BaseResidueClass) .**__xor__**(arg)
Logical XOR of residue class and *arg*.

Returns sieve.

21.1 Concrete classes

21.1.1 spannertools.Beam



`class spannertools.Beam (direction=None, overrides=None)`

A beam.

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'2")
>>> set_(staff).auto_beaming = False
>>> show(staff)
```



```
>>> beam = Beam()
>>> attach(beam, staff[:2])
>>> beam = Beam()
>>> attach(beam, staff[2:4])
>>> show(staff)
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
Selects components in spanner.
Returns selection.

`Beam.direction`
Gets direction of beam.
Returns up or down.

Special methods

`(Spanner).__contains__(expr)`
Is 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 Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

(Spanner) .**__getitem__**(*expr*)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(Spanner) .**__len__**()

Gets number of components in spanner.

Returns nonnegative integer.

(Spanner) .**__lt__**(*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

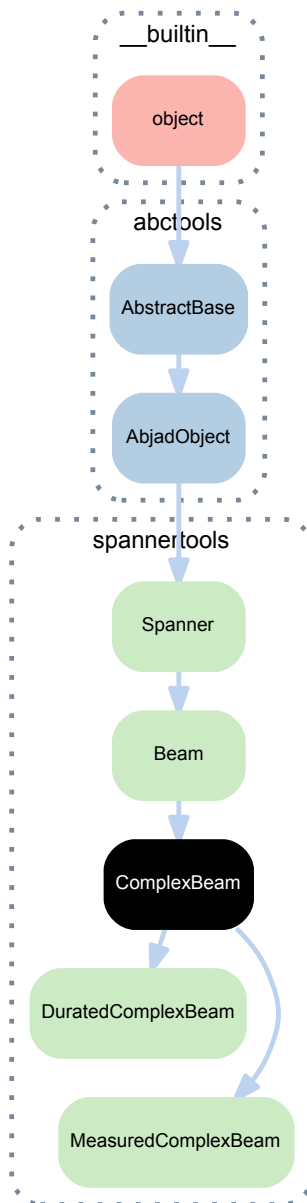
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

21.1.2 spannertools.ComplexBeam



class `spannertools.ComplexBeam` (*direction=None*, *isolated_nib_direction=False*, *over-rides=None*)

A complex beam.

```
>>> staff = Staff("c'16 e'16 r16 f'16 g'2")
>>> set_(staff).auto_beaming = False
>>> show(staff)
```



```
>>> beam = spannertools.ComplexBeam()
>>> attach(beam, staff[:4])
>>> show(staff)
```



Bases

- `spannertools.Beam`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
Selects components in spanner.
Returns selection.

`(Beam).direction`
Gets direction of beam.
Returns up or down.

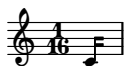
`ComplexBeam.isolated_nib_direction`
Gets directed treatment to apply to lone nibs.
Beams lone leaf and forces nib to the left:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=Left)
>>> attach(beam, measure)
>>> show(measure)
```



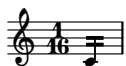
Beams lone leaf and forces nib to the right:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=Right)
>>> attach(beam, measure)
>>> show(measure)
```



Beams lone leaf and forces nibs both left and right:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=True)
>>> attach(beam, measure)
>>> show(measure)
```



Does not beam isolated_nib_direction leaf:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=False)
>>> attach(beam, measure)
>>> show(measure)
```



Set to left, right, true or false.

Ignores this setting when spanner contains more than one leaf.

Special methods

(Spanner) .**__contains__**(*expr*)

Is 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 Abjad object.

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

Returns string.

(Spanner) .**__getitem__**(*expr*)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(Spanner) .**__len__**()

Gets number of components in spanner.

Returns nonnegative integer.

(Spanner) .**__lt__**(*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

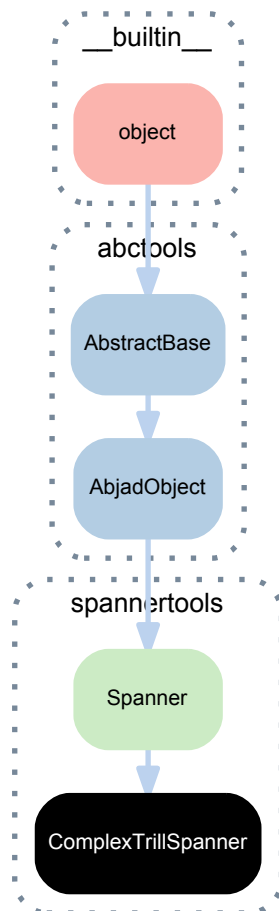
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

21.1.3 spannertools.ComplexTrillSpanner

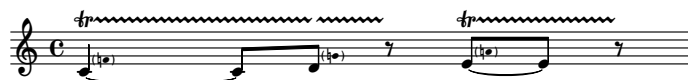


class `spannertools.ComplexTrillSpanner` (*overrides=None, interval=None*)
 A complex trill spanner.

```
>>> staff = Staff("c'4 ~ c'8 d'8 r8 e'8 ~ e'8 r8")
>>> show(staff)
```



```
>>> complex_trill = spannertools.ComplexTrillSpanner(
...     interval='P4',
... )
>>> attach(complex_trill, staff.select_leaves())
>>> show(staff)
```



Allows for specifying a trill pitch via a named interval.

Avoids silences.

Restarts the trill on every new pitched logical tie.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`

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

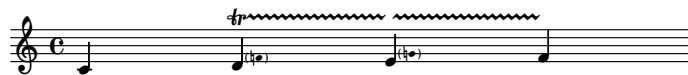
Read-only properties

`(Spanner).components`
Selects components in spanner.

Returns selection.

`ComplexTrillSpanner.interval`
Gets optional interval of trill spanner.

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> interval = pitchtools.NamedInterval('m3')
>>> complex_trill = spannertools.ComplexTrillSpanner(
...     interval=interval)
>>> attach(complex_trill, staff[1:-1])
>>> show(staff)
```



```
>>> complex_trill.interval
NamedInterval('m3')
```

Special methods

`(Spanner).__contains__(expr)`
Is 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 Abjad object.

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

Returns string.

`(Spanner).__getitem__(expr)`
Gets item from spanner.

Returns component.

`(AbjadObject).__hash__()`
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

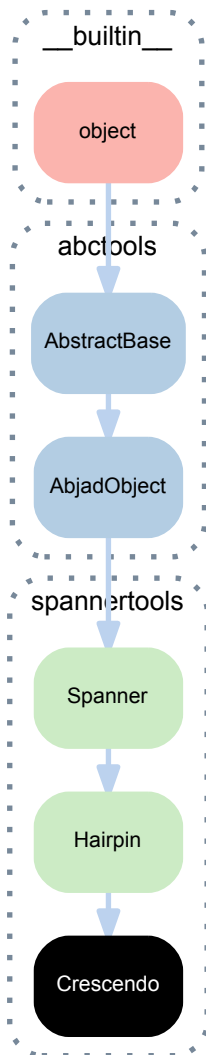
(`Spanner`) .`__len__`()
 Gets number of components in spanner.
 Returns nonnegative integer.

(`Spanner`) .`__lt__`(*expr*)
 Is true when spanner is less than *expr*. Otherwise false.
 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.

21.1.4 spannertools.Crescendo



class `spannertools.Crescendo` (*direction=None, include_rests=False, overrides=None*)
 A crescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> crescendo = spannertools.Crescendo()
>>> attach(crescendo, staff[:])
>>> show(staff)
```



Bases

- `spannertools.Hairpin`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**
Selects components in spanner.
Returns selection.

Crescendo **.descriptor**
Gets descriptor of crescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> crescendo = Crescendo()
>>> attach(crescendo, staff[:])
>>> show(staff)
```



```
>>> crescendo.descriptor
'<'
```

Returns string.

Crescendo **.direction**
Gets direction of crescendo.

Positions crescendo above staff:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> crescendo = Crescendo(direction=Up)
>>> attach(crescendo, staff[:])
>>> show(staff)
```



Returns up, down or none.

Crescendo **.include_rests**
Gets include-rests flag of crescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> crescendo = Crescendo(include_rests=True)
>>> attach(crescendo, staff[:])
>>> show(staff)
```



Returns boolean.

`Crescendo.shape_string`

Gets shape string of crescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> crescendo = Crescendo()
>>> attach(crescendo, staff[:])
>>> show(staff)
```



```
>>> crescendo.shape_string
'<'
```

Returns string.

`Crescendo.start_dynamic`

Gets start dynamic of crescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> crescendo = Crescendo()
>>> attach(crescendo, staff[:])
>>> show(staff)
```



```
>>> crescendo.start_dynamic is None
True
```

Returns dynamic or none.

`Crescendo.stop_dynamic`

Gets stop dynamic of crescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> crescendo = Crescendo()
>>> attach(crescendo, staff[:])
>>> show(staff)
```



```
>>> crescendo.stop_dynamic is None
True
```

Returns dynamic or none.

Special methods

`(Spanner).__contains__(expr)`

Is 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 Abjad object.

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

Returns string.

(Spanner) .**__getitem__**(*expr*)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(Spanner) .**__len__**()

Gets number of components in spanner.

Returns nonnegative integer.

(Spanner) .**__lt__**(*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

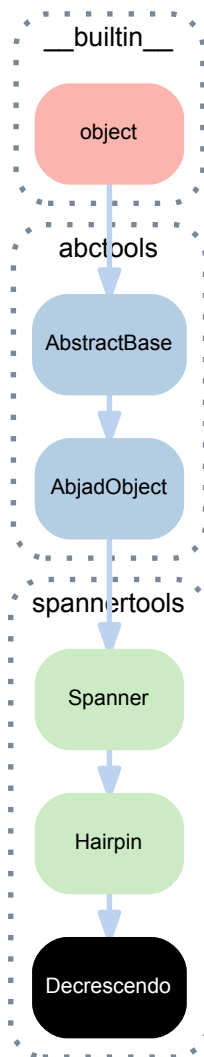
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

21.1.5 spannertools.Decrescendo



class `spannertools.Decrescendo` (*direction=None, include_rests=False, overrides=None*)
 A decrescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> decrescendo = spannertools.Decrescendo()
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



Bases

- `spannertools.Hairpin`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**

Selects components in spanner.

Returns selection.

Decrescendo **.descriptor**

Gets descriptor of decrescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> decrescendo = Decrescendo()
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



```
>>> decrescendo.descriptor
'>'
```

Returns string.

Decrescendo **.direction**

Gets direction of decrescendo.

Positions decrescendo above staff:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> decrescendo = Decrescendo(direction=Up)
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



Returns up, down or none.

Decrescendo **.include_rests**

Gets include-rests flag of decrescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> decrescendo = Decrescendo(include_rests=True)
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



Returns boolean.

Decrescendo **.shape_string**

Gets shape string of decrescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> decrescendo = Decrescendo()
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



```
>>> decrescendo.shape_string
'>'
```

Returns string.

Decrescendo.start_dynamic

Gets start dynamic of decrescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> decrescendo = Decrescendo()
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



```
>>> decrescendo.start_dynamic is None
True
```

Returns dynamic or none.

Decrescendo.stop_dynamic

Gets stop dynamic of decrescendo.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> decrescendo = Decrescendo()
>>> attach(decrescendo, staff[:])
>>> show(staff)
```



```
>>> decrescendo.stop_dynamic is None
True
```

Returns dynamic or none.

Special methods

(Spanner) .**__contains__**(*expr*)

Is 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 Abjad object.

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

Returns string.

(Spanner) .**__getitem__**(*expr*)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(Spanner) .**__len__**()

Gets number of components in spanner.

Returns nonnegative integer.

(Spanner) .**__lt__**(*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

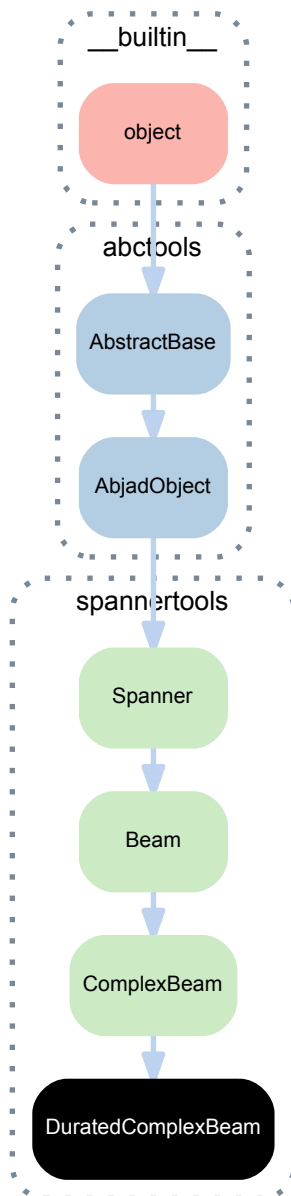
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

21.1.6 spannertools.DuratedComplexBeam



```

class spannertools.DuratedComplexBeam(direction=None, durations=None,
                                       isolated_nib_direction=False,
                                       nibs_towards_nonbeamable_components=True,
                                       overrides=None, span_beam_count=1)

```

A durated complex beam.

```

>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> set_(staff).auto_beaming = False
>>> show(staff)

```



```

>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = spannertools.DuratedComplexBeam(
...     durations=durations,
...     span_beam_count=1,
... )
>>> attach(beam, staff[:])
>>> show(staff)

```



Beams all beamable leaves in spanner explicitly.

Groups leaves in spanner according to *durations*.

Spans leaves between duration groups according to *span_beam_count*.

Bases

- `spannertools.ComplexBeam`
- `spannertools.Beam`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**

Selects components in spanner.

Returns selection.

(Beam) **.direction**

Gets direction of beam.

Returns up or down.

DuratedComplexBeam **.durations**

Gets durations of leaf groups in spanner.

```
>>> 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[:])
>>> show(staff)
```



```
>>> beam.durations
(Duration(1, 8), Duration(1, 8))
```

Returns tuple of durations or none.

(ComplexBeam) **.isolated_nib_direction**

Gets directed treatment to apply to lone nibs.

Beams lone leaf and forces nib to the left:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=Left)
>>> attach(beam, measure)
>>> show(measure)
```



Beams lone leaf and forces nib to the right:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=Right)
>>> attach(beam, measure)
>>> show(measure)
```



Beams lone leaf and forces nibs both left and right:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=True)
>>> attach(beam, measure)
>>> show(measure)
```



Does not beam isolated_nib_direction leaf:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=False)
>>> attach(beam, measure)
>>> show(measure)
```



Set to left, right, true or false.

Ignores this setting when spanner contains more than one leaf.

DuratedComplexBeam.nibs_towards_nonbeamable_components

Is true when when spanner should render nibs pointing towards nonbeamable components included in spanner. Otherwise false.

Does not draw nibs towards nonbeamable components:

```
>>> staff = Staff("c'16 d'16 r4 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 4), Duration(1, 8)]
>>> beam = spannertools.DuratedComplexBeam(
...     durations=durations,
...     nibs_towards_nonbeamable_components=False
... )
>>> attach(beam, staff[:])
>>> show(staff)
```



Do draw nibs towards nonbeamable components:

```
>>> staff = Staff("c'16 d'16 r4 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 4), Duration(1, 8)]
>>> beam = spannertools.DuratedComplexBeam(
...     durations=durations,
...     nibs_towards_nonbeamable_components=True
... )
>>> attach(beam, staff[:])
>>> show(staff)
```



Defaults to true.

Returns boolean.

DuratedComplexBeam.**span_beam_count**

Gets span beam count of spanner.

Creates a single span beam between adjacent groups in spanner:

```
>>> staff = Staff("c'32 d'32 e'32 f'32")
>>> durations = [Duration(1, 16), Duration(1, 16)]
>>> beam = spannertools.DuratedComplexBeam(
...     durations=durations,
...     span_beam_count=1,
... )
>>> attach(beam, staff[:])
>>> show(staff)
```



```
>>> beam.span_beam_count
1
```

Creates a double span beam between adjacent groups in spanner:

```
>>> staff = Staff("c'32 d'32 e'32 f'32")
>>> durations = [Duration(1, 16), Duration(1, 16)]
>>> beam = spannertools.DuratedComplexBeam(
...     durations=durations,
...     span_beam_count=2,
... )
>>> attach(beam, staff[:])
>>> show(staff)
```



```
>>> beam.span_beam_count
2
```

Creates no span beam between adjacent groups in spanner:

```
>>> staff = Staff("c'32 d'32 e'32 f'32")
>>> durations = [Duration(1, 16), Duration(1, 16)]
>>> beam = spannertools.DuratedComplexBeam(
...     durations=durations,
...     span_beam_count=0,
... )
>>> attach(beam, staff[:])
>>> show(staff)
```



```
>>> beam.span_beam_count
0
```

Defaults to 1.

Returns nonnegative integer.

Special methods

(Spanner).**__contains__**(*expr*)

Is 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 Abjad object.

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

Returns string.

(Spanner) .**__getitem__**(*expr*)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(Spanner) .**__len__**()

Gets number of components in spanner.

Returns nonnegative integer.

(Spanner) .**__lt__**(*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

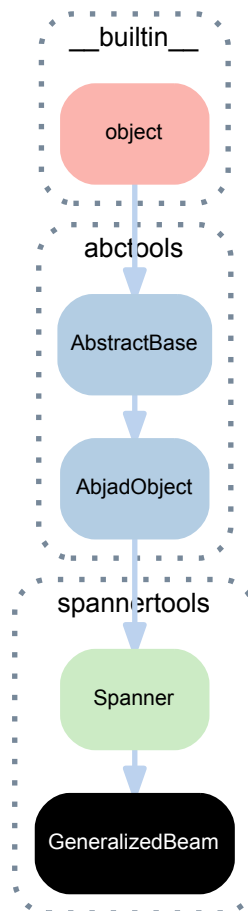
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

21.1.7 spannertools.GeneralizedBeam



```
class spannertools.GeneralizedBeam(durations=None, include_long_duration_notes=False,
                                   include_long_duration_rests=False, isolated_nib_direction=None, use_stemlets=False, vertical_direction=None)
```

A generalized beam.

```
>>> staff = Staff("r4 c'8 d'16 e'16 r8 fs'8 g'4")
>>> set_(staff).auto_beaming = False
>>> show(staff)
```



```
>>> beam = spannertools.GeneralizedBeam()
>>> attach(beam, staff[:])
>>> show(staff)
```



```
>>> staff = Staff("r4 c'8 d'16 e'16 r8 fs'8 g'4")
>>> set_(staff).auto_beaming = False
>>> show(staff)
```



```
>>> beam = spannertools.GeneralizedBeam(
...     isolated_nib_direction=Right,
... )
```

```
>>> attach(beam, staff[:])
>>> show(staff)
```



```
>>> staff = Staff("r4 c'8 d'16 e'16 r8 fs'8 g'4")
>>> set_(staff).auto_beaming = False
>>> show(staff)
```



```
>>> beam = spannertools.GeneralizedBeam(
...     use_stemlets=True,
... )
>>> attach(beam, staff[:])
>>> show(staff)
```



Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
Selects components in spanner.
Returns selection.

`GeneralizedBeam.durations`
Durations to use for span-beam groupings.

`GeneralizedBeam.include_long_duration_notes`
True if beam includes long duration notes, otherwise false.

`GeneralizedBeam.include_long_duration_rests`
True if beam includes long duration rests, otherwise false.

`GeneralizedBeam.isolated_nib_direction`
Direction of isolated nibs.

`GeneralizedBeam.use_stemlets`
True if beam uses stemlets, otherwise false.

`GeneralizedBeam.vertical_direction`
Vertical direction of the beam.

Special methods

`(Spanner).__contains__(expr)`
Is 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 Abjad object.

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

Returns string.

(Spanner) .**__getitem__** (expr)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__** ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(Spanner) .**__len__** ()

Gets number of components in spanner.

Returns nonnegative integer.

(Spanner) .**__lt__** (expr)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

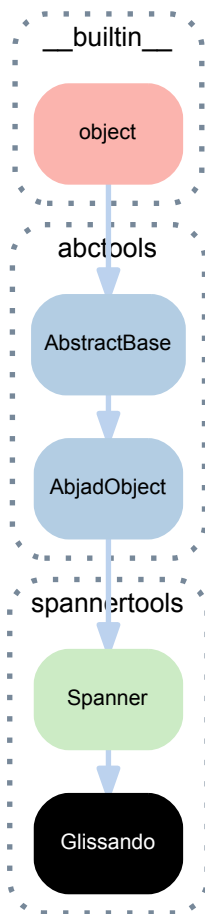
Returns boolean.

(AbjadObject) .**__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

21.1.8 spannertools.Glissando



class `spannertools.Glissando` (*overrides=None*)
 A glissando.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> glissando = spannertools.Glissando()
>>> attach(glissando, staff[:])
>>> show(staff)
```



Formats nonlast leaves in spanner with LilyPond `\glissando` command.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

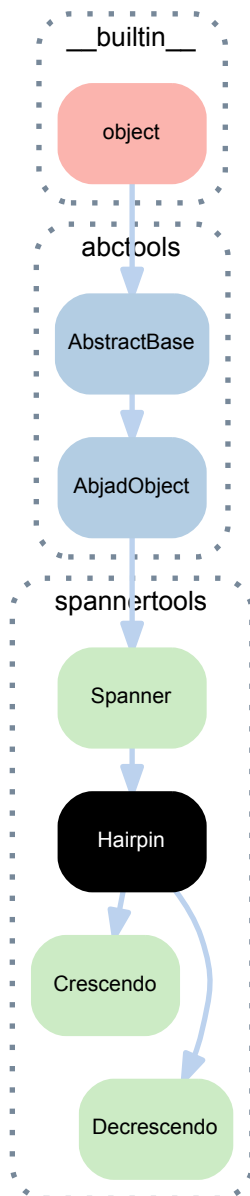
Read-only properties

(`Spanner`) . **components**
 Selects components in spanner.
 Returns selection.

Special methods

- (*Spanner*) .__contains__(*expr*)
Is 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 Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.
- (*Spanner*) .__getitem__(*expr*)
Gets item from spanner.
Returns component.
- (*AbjadObject*) .__hash__()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if __eq__ changes.
Returns integer.
- (*Spanner*) .__len__()
Gets number of components in spanner.
Returns nonnegative integer.
- (*Spanner*) .__lt__(*expr*)
Is true when spanner is less than *expr*. Otherwise false.
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.

21.1.9 spannertools.Hairpin



class spannertools.**Hairpin** (*descriptor='<', direction=None, include_rests=False, overrides=None*)

A hairpin.

```

>>> 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)
  
```



Bases

- spannertools.Spanner

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

Read-only properties

`(Spanner).components`

Selects components in spanner.

Returns selection.

`Hairpin.descriptor`

Gets descriptor of hairpin.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> show(staff)
```



```
>>> hairpin.descriptor
'p < f'
```

Returns string.

`Hairpin.direction`

Gets direction of hairpin.

Positions hairpin above staff:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> hairpin = spannertools.Hairpin(
...     descriptor='p < f',
...     direction=Up,
... )
>>> attach(hairpin, staff[:])
>>> show(staff)
```



Returns up, down or none.

`Hairpin.include_rests`

Gets include-rests flag of hairpin.

```
>>> 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)
```



Returns boolean.

`Hairpin.shape_string`

Gets shape string of hairpin.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> show(staff)
```



```
>>> hairpin.shape_string
'<'
```

Returns string.

`Hairpin.start_dynamic`

Gets start dynamic string of hairpin.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> show(staff)
```



```
>>> hairpin.start_dynamic
Dynamic(name='p')
```

Returns dynamic or none.

`Hairpin.stop_dynamic`

Gets stop dynamic string of hairpin.

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> hairpin = spannertools.Hairpin(descriptor='p < f')
>>> attach(hairpin, staff[:])
>>> show(staff)
```



```
>>> hairpin.stop_dynamic
Dynamic(name='f')
```

Returns dynamic or none.

Special methods

`(Spanner).__contains__(expr)`

Is 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 Abjad object.

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

Returns string.

(Spanner) .**__getitem__** (*expr*)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__** ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(Spanner) .**__len__** ()

Gets number of components in spanner.

Returns nonnegative integer.

(Spanner) .**__lt__** (*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

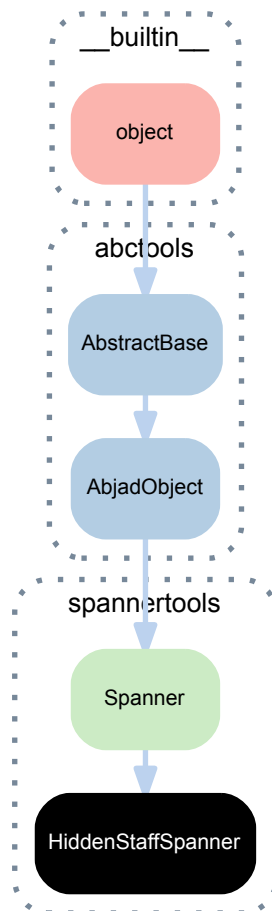
Returns boolean.

(AbjadObject) .**__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

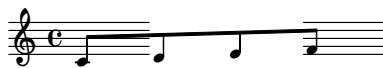
21.1.10 spannertools.HiddenStaffSpanner



class spannertools.**HiddenStaffSpanner** (*overrides=None*)
 A hidden staff spanner.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.HiddenStaffSpanner()
>>> attach(spanner, staff[1:3])
>>> show(staff)
  
```



Formats LilyPond `\stopStaff` before first leaf in spanner.

Formats LilyPond `\startStaff` command after last leaf in spanner.

Bases

- spannertools.Spanner
- abctools.AbjadObject
- abctools.AbjadObject.AbstractBase
- __builtin__.object

Read-only properties

(Spanner).**components**
 Selects components in spanner.

Returns selection.

Special methods

(*Spanner*) .__contains__(*expr*)

Is 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 Abjad object.

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

Returns string.

(*Spanner*) .__getitem__(*expr*)

Gets item from spanner.

Returns component.

(*AbjadObject*) .__hash__()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if __eq__ changes.

Returns integer.

(*Spanner*) .__len__()

Gets number of components in spanner.

Returns nonnegative integer.

(*Spanner*) .__lt__(*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

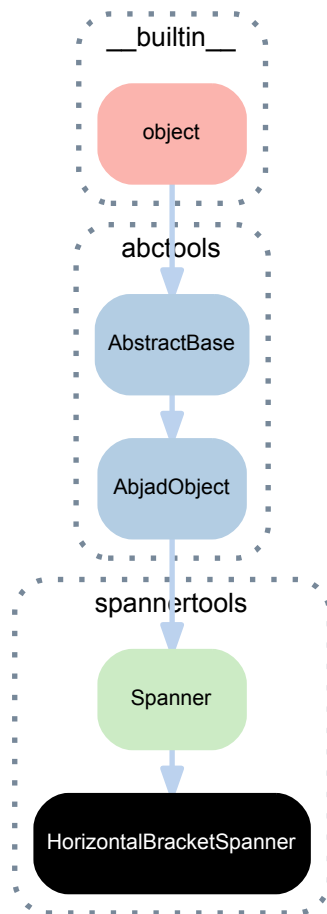
Returns boolean.

(*AbjadObject*) .__repr__()

Gets interpreter representation of Abjad object.

Returns string.

21.1.11 spannertools.HorizontalBracketSpanner

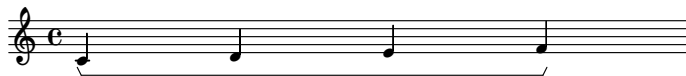


class spannertools.**HorizontalBracketSpanner** (*overrides=None*)
 A horizontal bracket spanner.

```

>>> voice = Voice("c'4 d'4 e'4 f'4")
>>> voice.consists_commands.append('Horizontal_bracket_engraver')
>>> spanner = spannertools.HorizontalBracketSpanner()
>>> attach(spanner, voice[:])
>>> show(voice)

```



Formats LilyPond `\startGroup` command on first leaf in spanner.

Formats LilyPond `\stopGroup` command on last leaf in spanner.

Bases

- spannertools.Spanner
- abctools.AbjadObject
- abctools.AbjadObject.AbstractBase
- __builtin__.object

Read-only properties

(*Spanner*) . **components**
Selects components in spanner.
Returns selection.

Special methods

(*Spanner*) . **__contains__** (*expr*)
Is 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 Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

(*Spanner*) . **__getitem__** (*expr*)
Gets item from spanner.
Returns component.

(*AbjadObject*) . **__hash__** ()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
Returns integer.

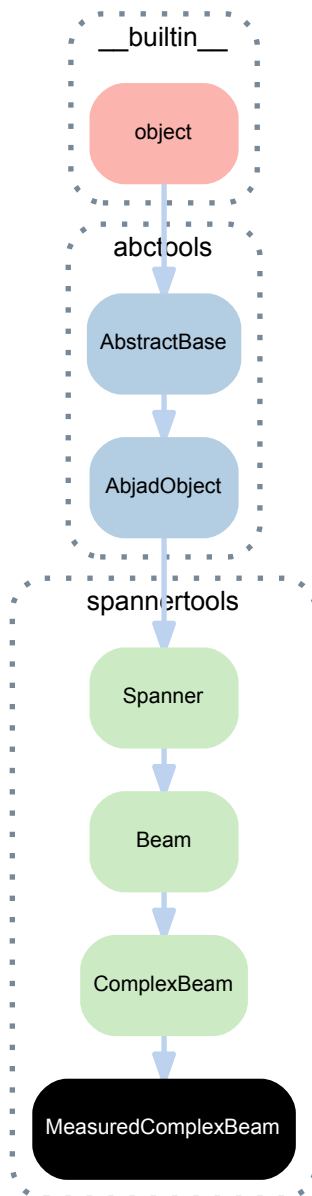
(*Spanner*) . **__len__** ()
Gets number of components in spanner.
Returns nonnegative integer.

(*Spanner*) . **__lt__** (*expr*)
Is true when spanner is less than *expr*. Otherwise false.
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.

21.1.12 spannertools.MeasuredComplexBeam



class `spannertools.MeasuredComplexBeam` (*direction=None*, *isolated_nib_direction=False*, *overrides=None*, *span_beam_count=1*)

A measured complex beam.

```

>>> staff = Staff()
>>> staff.append(Measure((2, 16), "c'16 d'16"))
>>> staff.append(Measure((2, 16), "e'16 f'16"))
>>> set_(staff).auto_beaming = False
>>> 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_count* beam between measures.

Bases

- `spannertools.ComplexBeam`
- `spannertools.Beam`
- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**

Selects components in spanner.

Returns selection.

(Beam) **.direction**

Gets direction of beam.

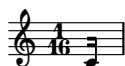
Returns up or down.

(ComplexBeam) **.isolated_nib_direction**

Gets directed treatment to apply to lone nibs.

Beams lone leaf and forces nib to the left:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=Left)
>>> attach(beam, measure)
>>> show(measure)
```



Beams lone leaf and forces nib to the right:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=Right)
>>> attach(beam, measure)
>>> show(measure)
```



Beams lone leaf and forces nibs both left and right:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=True)
>>> attach(beam, measure)
>>> show(measure)
```



Does not beam isolated_nib_direction leaf:

```
>>> measure = Measure((1, 16), "c'16")
>>> beam = spannertools.ComplexBeam(isolated_nib_direction=False)
>>> attach(beam, measure)
>>> show(measure)
```



Set to left, right, true or false.

Ignores this setting when spanner contains more than one leaf.

`MeasuredComplexBeam.span_beam_count`

Gets number of span beams between adjacent measures.

Use one span beam between measures:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 32), "c'32 d'32"))
>>> staff.append(Measure((2, 32), "e'32 f'32"))
>>> beam = spannertools.MeasuredComplexBeam(span_beam_count=1)
>>> attach(beam, staff.select_leaves())
>>> show(staff)
```



```
>>> beam.span_beam_count
1
```

Use two span beams between measures:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 32), "c'32 d'32"))
>>> staff.append(Measure((2, 32), "e'32 f'32"))
>>> beam = spannertools.MeasuredComplexBeam(span_beam_count=2)
>>> attach(beam, staff.select_leaves())
>>> show(staff)
```



```
>>> beam.span_beam_count
2
```

Returns nonnegative integer or none.

Special methods

`(Spanner).__contains__(expr)`

Is 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 Abjad object.

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

Returns string.

(*Spanner*) .__**getitem**__ (*expr*)

Gets item from spanner.

Returns component.

(*AbjadObject*) .__**hash**__ ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(*Spanner*) .__**len**__ ()

Gets number of components in spanner.

Returns nonnegative integer.

(*Spanner*) .__**lt**__ (*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

(*AbjadObject*) .__**ne**__ (*expr*)

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

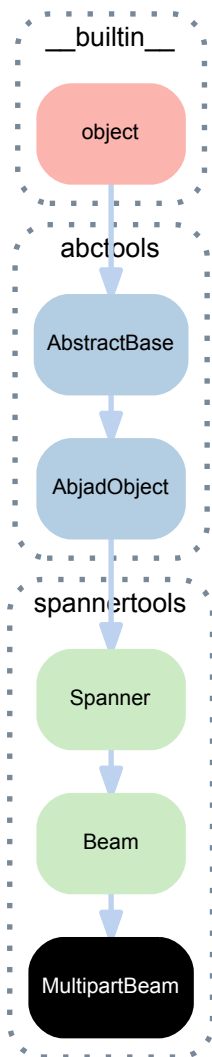
Returns boolean.

(*AbjadObject*) .__**repr**__ ()

Gets interpreter representation of Abjad object.

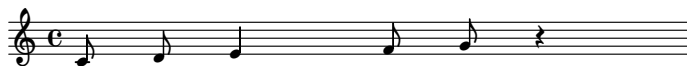
Returns string.

21.1.13 spannertools.MultipartBeam

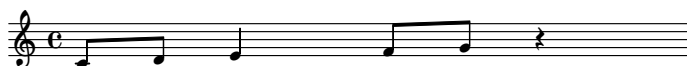


class spannertools.**MultipartBeam** (*direction=None, overrides=None*)
 A multipart beam.

```
>>> staff = Staff("c'8 d'8 e'4 f'8 g'8 r4")
>>> set_(staff).auto_beaming = False
>>> show(staff)
```



```
>>> beam = spannertools.MultipartBeam()
>>> attach(beam, staff[:])
>>> show(staff)
```



Avoids rests.

Avoids large-duration notes.

Bases

- spannertools.Beam

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
Selects components in spanner.
Returns selection.

`(Beam).direction`
Gets direction of beam.
Returns up or down.

Special methods

`(Spanner).__contains__(expr)`
Is 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 Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(Spanner).__getitem__(expr)`
Gets item from spanner.
Returns component.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
Returns integer.

`(Spanner).__len__()`
Gets number of components in spanner.
Returns nonnegative integer.

`(Spanner).__lt__(expr)`
Is true when spanner is less than *expr*. Otherwise false.
Trivial comparison to allow doctests to work.
Returns boolean.

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

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

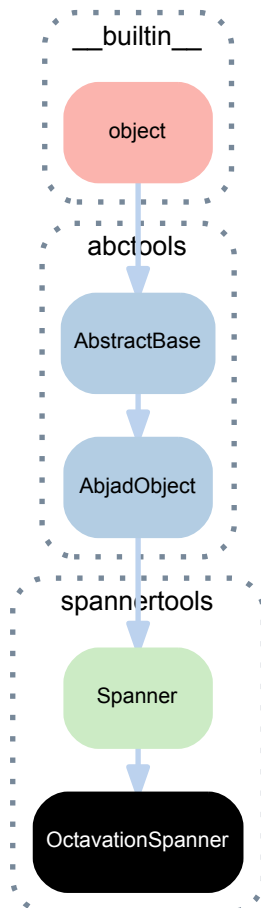
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

21.1.14 spannertools.OctavationSpanner



class spannertools.**OctavationSpanner** (*overrides=None, start=1, stop=0*)

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

Read-only properties

`(Spanner).components`
Selects components in spanner.

Returns selection.

`OctavationSpanner.start`
Gets octavation start.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.OctavationSpanner(start=1)
>>> attach(spanner, staff[:])
>>> show(staff)
```

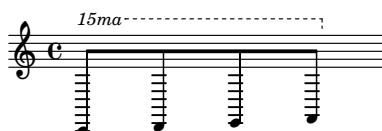


```
>>> spanner.start
1
```

Returns integer or none.

`OctavationSpanner.stop`
Gets octavation stop.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.OctavationSpanner(start=2, stop=1)
>>> attach(spanner, staff[:])
>>> show(staff)
```



```
>>> spanner.stop
1
```

Returns integer or none.

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.

Special methods

(Spanner) .**__contains__**(*expr*)

Is 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 Abjad object.

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

Returns string.

(Spanner) .**__getitem__**(*expr*)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(Spanner) .**__len__**()

Gets number of components in spanner.

Returns nonnegative integer.

(Spanner) .**__lt__**(*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

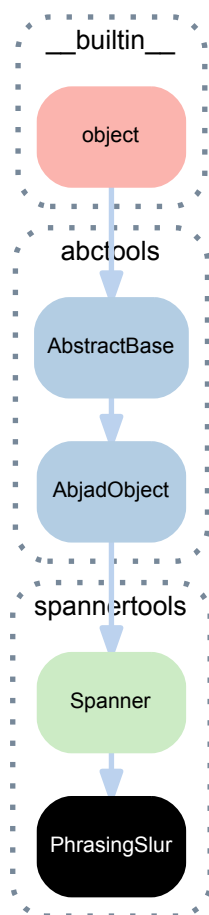
(AbjadObject) .**__ne__**(*expr*)

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

Returns boolean.

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

21.1.15 spannertools.PhrasingSlur



class `spannertools.PhrasingSlur` (*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)
  
```



Formats LilyPond \ (command on first leaf in spanner.

Formats LilyPond \) comand on last leaf in spanner.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Spanner) . **components**

Selects components in spanner.

Returns selection.

PhrasingSlur . **direction**

Gets direction of phrasing slur.

Positions phrasing slur above staff:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.PhrasingSlur(direction=Up)
>>> attach(slur, staff[:])
>>> show(staff)
```



Positions phrasing slur below staff:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.PhrasingSlur(direction=Down)
>>> attach(slur, staff[:])
>>> show(staff)
```



Positions phrasing slur according to LilyPond defaults:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.PhrasingSlur(direction=None)
>>> attach(slur, staff[:])
>>> show(staff)
```



Returns up, down or none.

Special methods

(Spanner) . **__contains__** (*expr*)

Is 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 Abjad object.

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

Returns string.

(*Spanner*) . **__getitem__** (*expr*)

Gets item from spanner.

Returns component.

(*AbjadObject*) . **__hash__** ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(*Spanner*) . **__len__** ()

Gets number of components in spanner.

Returns nonnegative integer.

(*Spanner*) . **__lt__** (*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

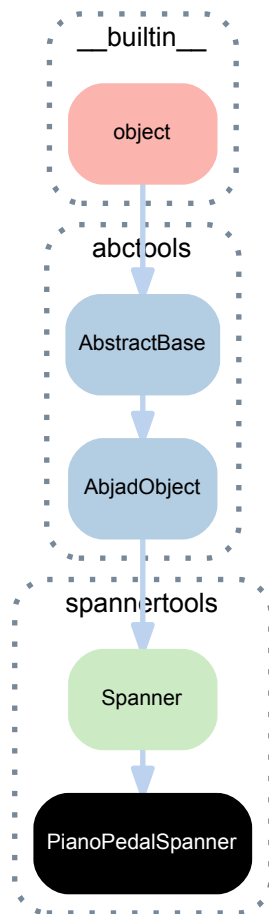
Returns boolean.

(*AbjadObject*) . **__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

21.1.16 spannertools.PianoPedalSpanner



class `spannertools.PianoPedalSpanner` (*kind='sustain', overrides=None, style='mixed'*)
 A piano pedal spanner.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> pedal = spannertools.PianoPedalSpanner()
>>> attach(pedal, staff[:])
>>> show(staff)
  
```



Formats LilyPond `\sustainOn`, `\sosenutoOn` or `\unaCora` on first leaf in spanner.

Formats LilyPond `\sustainOff`, `\sostenutoOff` or `\treCorde` on last leaf in spanner.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Spanner).**components**

Selects components in spanner.

Returns selection.

PianoPedalSpanner.**kind**

Gets kind of piano pedal spanner.

Sustain pedal:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(kind='sustain')
>>> attach(spanner, staff[:])
>>> show(staff)
```



```
>>> spanner.kind
'sustain'
```

Sostenuto pedal:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(kind='sostenuto')
>>> attach(spanner, staff[:])
>>> show(staff)
```



```
>>> spanner.kind
'sostenuto'
```

Una corda / tre corde pedal:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(kind='corda')
>>> attach(spanner, staff[:])
>>> show(staff)
```



```
>>> spanner.kind
'corda'
```

Returns 'sustain', 'sostenuto' or 'corda'.

PianoPedalSpanner.**style**

Gets style of piano pedal spanner.

Mixed style:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(style='mixed')
>>> attach(spanner, staff[:])
>>> show(staff)
```




```
>>> spanner.style
'mixed'
```

Bracket style:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(style='bracket')
>>> attach(spanner, staff[:])
>>> show(staff)
```



```
>>> spanner.style
'bracket'
```

Text style:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(style='text')
>>> attach(spanner, staff[:])
>>> show(staff)
```



```
>>> spanner.style
'text'
```

Returns 'mixed', 'bracket' or 'text'.

Special methods

(Spanner) .__contains__ (expr)

Is 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 Abjad object.

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

Returns string.

(Spanner) .__getitem__ (expr)

Gets item from spanner.

Returns component.

(AbjadObject) .__hash__ ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

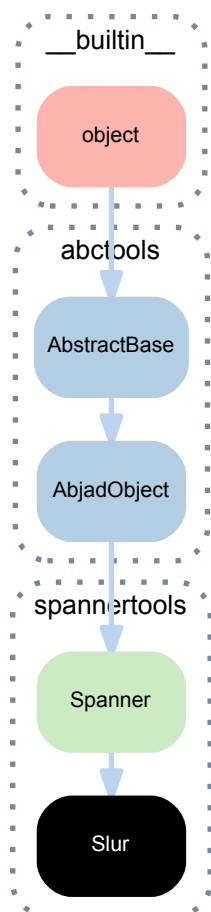
(*Spanner*) .__len__()
 Gets number of components in *spanner*.
 Returns nonnegative integer.

(*Spanner*) .__lt__(*expr*)
 Is true when *spanner* is less than *expr*. Otherwise false.
 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.

21.1.17 spannertools.Slur



class `spannertools.Slur` (*direction=None, overrides=None*)
 A slur.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.Slur()
>>> attach(slur, staff[:])
>>> show(staff)
  
```



Formats LilyPond (command on first leaf in spanner.

Formats LilyPond) command on last leaf in spanner.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Spanner) .**components**

Selects components in spanner.

Returns selection.

Slur .**direction**

Gets direction of slur.

Forces slur above staff:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.Slur(direction=Up)
>>> attach(slur, staff[:])
>>> show(staff)
```



Forces slur below staff:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.Slur(direction=Down)
>>> attach(slur, staff[:])
>>> show(staff)
```



Positions slur according to LilyPond defaults:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.Slur(direction=None)
>>> attach(slur, staff[:])
>>> show(staff)
```

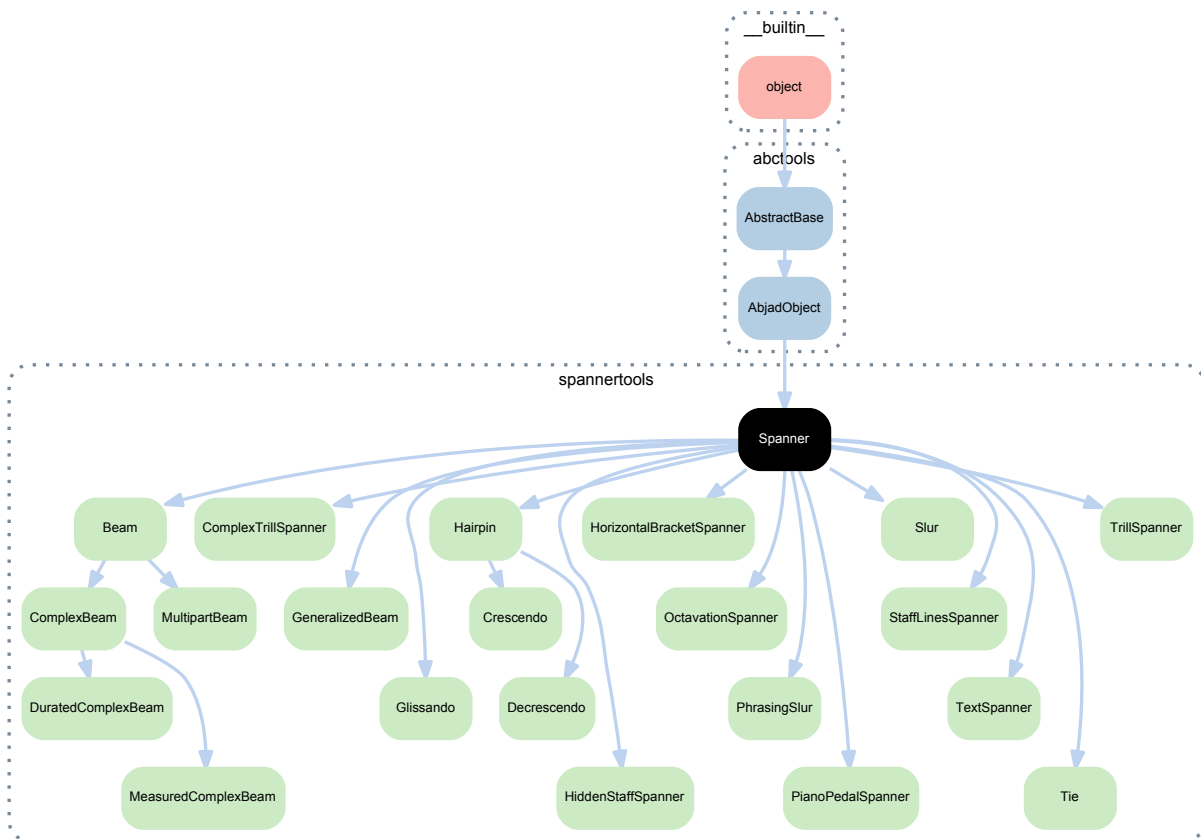


Returns up, down or none.

Special methods

- (*Spanner*) .**__contains__**(*expr*)
Is 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 Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.
- (*Spanner*) .**__getitem__**(*expr*)
Gets item from spanner.
Returns component.
- (*AbjadObject*) .**__hash__**()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if **__eq__** changes.
Returns integer.
- (*Spanner*) .**__len__**()
Gets number of components in spanner.
Returns nonnegative integer.
- (*Spanner*) .**__lt__**(*expr*)
Is true when spanner is less than *expr*. Otherwise false.
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.

21.1.18 spannertools.Spanner



class `spannertools.Spanner` (*overrides=None*)

Any type of object that stretches horizontally and encompasses some number of score components.

Examples include beams, slurs, hairpins and trills.

Bases

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

Read-only properties

`Spanner.components`

Selects components in spanner.

Returns selection.

Special methods

`Spanner.__contains__(expr)`

Is 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 Abjad object.

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

Returns string.

Spanner .**__getitem__** (*expr*)

Gets item from spanner.

Returns component.

(AbjadObject) .**__hash__** ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

Spanner .**__len__** ()

Gets number of components in spanner.

Returns nonnegative integer.

Spanner .**__lt__** (*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

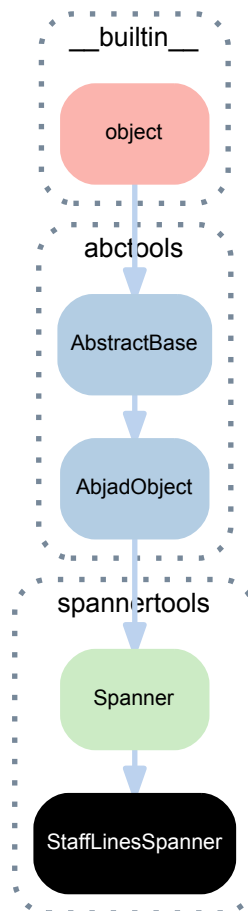
Returns boolean.

(AbjadObject) .**__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

21.1.19 spannertools.StaffLinesSpanner



class `spannertools.StaffLinesSpanner` (*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[1:3])
>>> show(staff)
  
```



Stops and restarts staff on first leaf in spanner.

Overrides `line-count` attribute of LilyPond `Staff.StaffSymbol` grob on first leaf in spanner.

Stops and restarts staff on last leaf in spanner.

Reverts `line-count` attribute of LilyPond `Staff.StaffSymbol` grob on last leaf in spanner.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
Selects components in spanner.
Returns selection.

`StaffLinesSpanner.lines`
Gets line of staff lines spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.StaffLinesSpanner(lines=1)
>>> attach(spanner, staff[1:3])
>>> show(staff)
```



```
>>> spanner.lines
1
```

Returns nonnegative integer.

Special methods

`(Spanner).__contains__(expr)`
Is 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 Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

`(Spanner).__getitem__(expr)`
Gets item from spanner.
Returns component.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
Returns integer.

`(Spanner).__len__()`
Gets number of components in spanner.
Returns nonnegative integer.

(Spanner) .**__lt__**(*expr*)

Is true when spanner is less than *expr*. Otherwise false.

Trivial comparison to allow doctests to work.

Returns boolean.

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

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

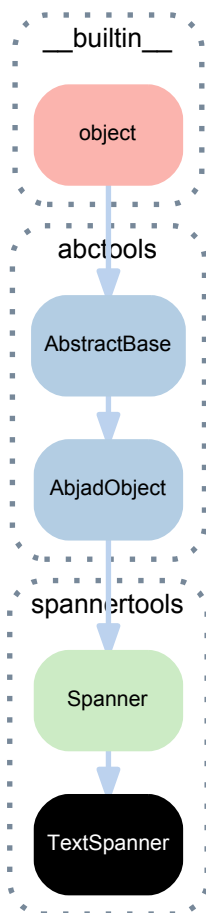
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

21.1.20 spannertools.TextSpanner



class spannertools.**TextSpanner** (*overrides=None*)

A text spanner.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> 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.bound_details__left__text = left_markup
>>> pair = schemetools.SchemePair(0, -1)
>>> markup_command = markuptools.MarkupCommand('draw-line', pair)
>>> right_markup = markuptools.Markup(markup_command)
>>> grob.bound_details__right__text = right_markup
  
```

```
>>> override(text_spanner).text_spanner.dash_fraction = 1
>>> attach(text_spanner, [staff])
>>> show(staff)
```



Formats LilyPond `\startTextSpan` command on first leaf in spanner.

Formats LilyPond `\stopTextSpan` command on last leaf in spanner.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
Selects components in spanner.
Returns selection.

Special methods

`(Spanner).__contains__(expr)`
Is 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 Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

`(Spanner).__getitem__(expr)`
Gets item from spanner.
Returns component.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
Returns integer.

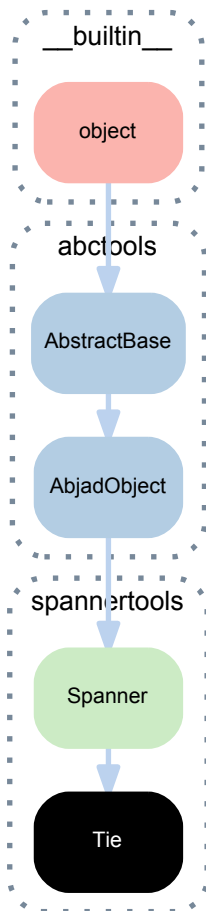
`(Spanner) .__len__()`
 Gets number of components in spanner.
 Returns nonnegative integer.

`(Spanner) .__lt__(expr)`
 Is true when spanner is less than *expr*. Otherwise false.
 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.

21.1.21 spannertools.Tie



class `spannertools.Tie` (*direction=None, overrides=None*)
 A collection of consecutive ties.

```

>>> staff = Staff("c'8 c'8 c'8 c'8")
>>> tie = Tie()
>>> attach(tie, staff[:])
>>> show(staff)
  
```



Formats LilyPond `~` command on nonlast leaves in spanner.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(Spanner) **.components**

Selects components in spanner.

Returns selection.

Tie **.direction**

Gets direction of ties in spanner.

Forces ties up:

```
>>> staff = Staff("c'8 c'8 c'8 c'8")
>>> tie = Tie(direction=Up)
>>> attach(tie, staff[:])
>>> show(staff)
```



```
>>> tie.direction
'^'
```

Forces ties down:

```
>>> staff = Staff("c'8 c'8 c'8 c'8")
>>> tie = Tie(direction=Down)
>>> attach(tie, staff[:])
>>> show(staff)
```



```
>>> tie.direction
'_'
```

Positions ties according to LilyPond defaults:

```
>>> staff = Staff("c'8 c'8 c'8 c'8")
>>> tie = Tie(direction=None)
>>> attach(tie, staff[:])
>>> show(staff)
```



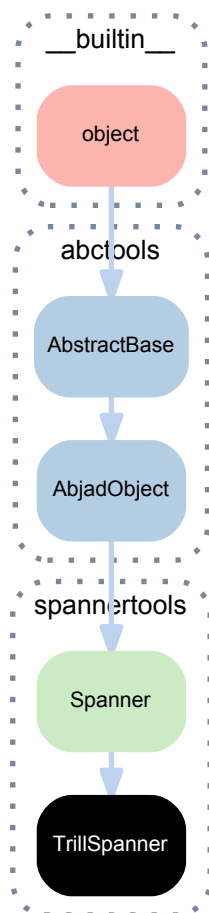
```
>>> tie.direction is None
True
```

Returns up, down or none.

Special methods

- (*Spanner*) .__contains__(*expr*)
Is 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 Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.
- (*Spanner*) .__getitem__(*expr*)
Gets item from spanner.
Returns component.
- (*AbjadObject*) .__hash__()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if __eq__ changes.
Returns integer.
- (*Spanner*) .__len__()
Gets number of components in spanner.
Returns nonnegative integer.
- (*Spanner*) .__lt__(*expr*)
Is true when spanner is less than *expr*. Otherwise false.
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.

21.1.22 spannertools.TrillSpanner



class `spannertools.TrillSpanner` (*overrides=None, pitch=None*)
 A trill spanner.

```

>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> trill = spannertools.TrillSpanner()
>>> attach(trill, staff[:])
>>> show(staff)
  
```



Formats LilyPond `\startTrillSpan` on first leaf in spanner.

Formats LilyPond `\stopTrillSpan` on last leaf in spanner.

Bases

- `spannertools.Spanner`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Spanner).components`
 Selects components in spanner.

Returns selection.

`TrillSpanner.pitch`

Gets optional pitch of trill spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> pitch = NamedPitch('C#4')
>>> trill = spannertools.TrillSpanner(pitch=pitch)
>>> attach(trill, staff[:2])
>>> show(staff)
```



```
>>> trill.pitch
NamedPitch("cs' ")
```

Formats LilyPond `\pitchedTrill` command on first leaf in spanner.

Returns named pitch or none.

`TrillSpanner.written_pitch`

Gets written pitch of trill spanner.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> pitch = NamedPitch('C#4')
>>> trill = spannertools.TrillSpanner(pitch=pitch)
>>> attach(trill, staff[:2])
>>> show(staff)
```



```
>>> trill.written_pitch
NamedPitch("cs' ")
```

Defined equal to *pitch*.

Returns named pitch or none.

Special methods

`(Spanner).__contains__(expr)`

Is 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 Abjad object.

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

Returns string.

(Spanner) . **__getitem__** (*expr*)
Gets item from spanner.
Returns component.

(AbjadObject) . **__hash__** ()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
Returns integer.

(Spanner) . **__len__** ()
Gets number of components in spanner.
Returns nonnegative integer.

(Spanner) . **__lt__** (*expr*)
Is true when spanner is less than *expr*. Otherwise false.
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.

21.2 Functions

21.2.1 `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.

21.2.2 `spannertools.make_dynamic_spanner_below_with_nib_at_right`

`spannertools.make_dynamic_spanner_below_with_nib_at_right(dynamic_text)`
Makes dynamic spanner below with nib at right.

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.make_dynamic_spanner_below_with_nib_at_right('mp')
>>> attach(spanner, staff[:])
>>> show(staff)
```




Returns text spanner.

21.2.3 `spannertools.make_solid_text_spanner_with_nib`

`spannertools.make_solid_text_spanner_with_nib` (*left_text*, *direction=Up*)

Makes solid text spanner with nib at right.

Solid text spanner forced above staff:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.make_solid_text_spanner_with_nib(
...     'foo',
...     direction=Up,
... )
>>> attach(spanner, staff[:])
>>> show(staff)
```



Solid text spanner forced below staff:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.make_solid_text_spanner_with_nib(
...     'foo',
...     direction=Down,
... )
>>> attach(spanner, staff[:])
>>> show(staff)
```



Returns text spanner.

22.1 Functions

22.1.1 `stringtools.add_terminal_newlines`

`stringtools.add_terminal_newlines` (*lines*)
Adds terminal newlines to *lines*.

```
>>> lines = ['first line', 'second line']
>>> stringtools.add_terminal_newlines(lines)
['first line\n', 'second line\n']
```

Does 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.

22.1.2 `stringtools.arg_to_bidirectional_direction_string`

`stringtools.arg_to_bidirectional_direction_string` (*arg*)
Changes *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'
```

Returns *arg* when *arg* is 'up' or 'down'.

Returns string or none.

22.1.3 `stringtools.arg_to_bidirectional_lilypond_symbol`

`stringtools.arg_to_bidirectional_lilypond_symbol` (*arg*)
Changes *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)
' _ '
```

Returns *arg* when *arg* is '^' or '_'.

Returns string or none.

22.1.4 stringtools.arg_to_tridirectional_direction_string

`stringtools.arg_to_tridirectional_direction_string(arg)`
Changes *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'
```

Returns none when *arg* is none.

Returns string or none.

22.1.5 stringtools.arg_to_tridirectional_lilypond_symbol

`stringtools.arg_to_tridirectional_lilypond_symbol(arg)`
Changes *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)
'_'
```

Returns none when *arg* is none.

Returns *arg* when *arg* is '^', '-' or '_'.

Returns string or none.

22.1.6 stringtools.arg_to_tridirectional_ordinal_constant

`stringtools.arg_to_tridirectional_ordinal_constant(arg)`
Changes *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
```

Returns *arg* when *arg* is *Up**, *Center* or *Down*.

Returns ordinal constant or none.

22.1.7 stringtools.capitalize_start

`stringtools.capitalize_start(string)`
Capitalizes *string*.

```
>>> string = 'violin I'
```

```
>>> stringtools.capitalize_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.

22.1.8 stringtools.delimit_words

`stringtools.delimit_words(string)`
Delimits words in *string*.

Delimits words:

```
>>> stringtools.delimit_words('scale degrees 4 and 5.')
['scale', 'degrees', '4', 'and', '5']
```

Delimits conjoined words:

```
>>> stringtools.delimit_words('scale degrees 4and5.')
['scale', 'degrees', '4', 'and', '5']
```

Delimits lower camel case:

```
>>> stringtools.delimit_words('scaleDegrees4and5.')
['scale', 'Degrees', '4', 'and', '5']
```

Delimits upper camel case:

```
>>> stringtools.delimit_words('ScaleDegrees4and 5.')
['Scale', 'Degrees', '4', 'and', '5']
```

Delimits dash case:

```
>>> stringtools.delimit_words('scale-degrees-4-and-5.')
['scale', 'degrees', '4', 'and', '5']
```

Delimits shout case:

```
>>> stringtools.delimit_words('SCALE_DEGREES_4_AND_5.')
['SCALE', 'DEGREES', '4', 'AND', '5']
```

Returns list.

22.1.9 stringtools.format_input_lines_as_doc_string

`stringtools.format_input_lines_as_doc_string(input_lines)`

Formats *input_lines* as doc string.

Formats expressions intelligently.

Treats blank lines intelligently.

Captures hash-suffixed line output.

Use when writing docstrings.

Example skipped because docstring goes crazy on example input.

22.1.10 stringtools.format_input_lines_as_regression_test

`stringtools.format_input_lines_as_regression_test(input_lines, tab_width=3)`

Formats *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 {
      \\n\\t\\times 2/3 {\\n\\t\\tc'8 [\\n\\t\\td'8\\n\\t\\te'8\\n\\t}\\n\\tf'8 ]\\n}"
    '''

```

Formats expressions intelligently.

Treats blank lines intelligently.

Removes line-final hash characters.

Use when writing tests.

Returns string.

22.1.11 stringtools.is_dash_case

`stringtools.is_dash_case(expr)`

Is true when *expr* is a string and is hyphen delimited lowercase.

```
>>> stringtools.is_dash_case('foo-bar')
True
```

Otherwise false:

```
>>> stringtools.is_dash_case('foo bar')
False
```

Returns boolean.

22.1.12 stringtools.is_dash_case_file_name

`stringtools.is_dash_case_file_name(expr)`

Is 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.

22.1.13 stringtools.is_lower_camel_case

`stringtools.is_lower_camel_case(expr)`

Is true when *expr* is a string and is lowercamelcase.

```
>>> stringtools.is_lower_camel_case('fooBar')
True
```

Otherwise false:

```
>>> stringtools.is_lower_camel_case('FooBar')
False
```

Returns boolean.

22.1.14 stringtools.is_snake_case

`stringtools.is_snake_case(expr)`

Is true when *expr* is a string and is underscore delimited lowercase.

```
>>> stringtools.is_snake_case('foo_bar')
True
```

Otherwise false:

```
>>> stringtools.is_snake_case('foo bar')
False
```

Returns boolean.

22.1.15 stringtools.is_snake_case_file_name

`stringtools.is_snake_case_file_name(expr)`

Is 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.

22.1.16 stringtools.is_snake_case_file_name_with_extension

`stringtools.is_snake_case_file_name_with_extension(expr)`

Is 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.

22.1.17 stringtools.is_snake_case_package_name

`stringtools.is_snake_case_package_name(expr)`

Is 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.

22.1.18 stringtools.is_space_delimited_lowercase

`stringtools.is_space_delimited_lowercase(expr)`
Is true when *expr* is a string and is space-delimited lowercase.

```
>>> stringtools.is_space_delimited_lowercase('foo bar')
True
```

Otherwise false:

```
>>> stringtools.is_space_delimited_lowercase('foo_bar')
False
```

Returns boolean.

22.1.19 stringtools.is_string

`stringtools.is_string(expr)`
Is true when *expr* is a string.

Compatible under both Python 2.7.x and 3.x.

22.1.20 stringtools.is_upper_camel_case

`stringtools.is_upper_camel_case(expr)`
Is true when *expr* is a string and is uppercamelcase.

```
>>> stringtools.is_upper_camel_case('FooBar')
True
```

Otherwise false:

```
>>> stringtools.is_upper_camel_case('fooBar')
False
```

Returns boolean.

22.1.21 stringtools.pluralize

`stringtools.pluralize(string, count=None)`
Pluralizes English *string*.

Changes terminal -y to -ies:

```
>>> stringtools.pluralize('catenary')
'catenaries'
```

Adds -es to terminal -s, -sh, -x and -z:

```
>>> stringtools.pluralize('brush')
'brushes'
```

Adds -s to all other strings:

```
>>> stringtools.pluralize('shape')
'shapes'
```

Returns string.

22.1.22 stringtools.snake_case_to_lower_camel_case

`stringtools.snake_case_to_lower_camel_case` (*string*)
Changes underscore-delimited lowercase *string* to lowercamelcase.

```
>>> string = 'bass_figure_alignment_positioning'
>>> stringtools.snake_case_to_lower_camel_case(string)
'bassFigureAlignmentPositioning'
```

Returns string.

22.1.23 stringtools.snake_case_to_upper_camel_case

`stringtools.snake_case_to_upper_camel_case` (*string*)
Changes underscore-delimited lowercase *string* to uppercamelcase.

```
>>> string = 'bass_figure_alignment_positioning'
>>> stringtools.snake_case_to_upper_camel_case(string)
'BassFigureAlignmentPositioning'
```

Returns string.

22.1.24 stringtools.space_delimited_lowercase_to_upper_camel_case

`stringtools.space_delimited_lowercase_to_upper_camel_case` (*string*)
Changes space-delimited lowercase *string* to uppercamelcase.

```
>>> string = 'bass figure alignment positioning'
>>> stringtools.space_delimited_lowercase_to_upper_camel_case(string)
'BassFigureAlignmentPositioning'
```

Returns string.

22.1.25 stringtools.strip_diacritics

`stringtools.strip_diacritics` (*binary_string*)
Strips diacritics from *binary_string*.

```
>>> binary_string = 'Dvořák'
```

```
>>> print(binary_string)
Dvořák
```

```
>>> stringtools.strip_diacritics(binary_string)
'Dvorak'
```

Returns ASCII string.

22.1.26 stringtools.to_accent_free_snake_case

`stringtools.to_accent_free_snake_case` (*string*)
Changes *string* to accent-free snake case.

```
>>> stringtools.to_accent_free_snake_case('Déjà vu')
'deja_vu'
```

Strips accents from accented characters.

Changes all punctuation (including spaces) to underscore.

Sets to lowercase.

Returns string.

22.1.27 stringtools.to_dash_case

`stringtools.to_dash_case(string)`

Changes *string* to dash case.

Changes words to dash case:

```
>>> stringtools.to_dash_case('scale degrees 4 and 5')
'scale-degrees-4-and-5'
```

Changes snake case to dash case:

```
>>> stringtools.to_dash_case('scale_degrees_4_and_5')
'scale-degrees-4-and-5'
```

Changes dash case to dash case:

```
>>> stringtools.to_dash_case('scale-degrees-4-and-5')
'scale-degrees-4-and-5'
```

Changes upper camel case to dash case:

```
>>> stringtools.to_dash_case('ScaleDegrees4And5')
'scale-degrees-4-and-5'
```

Returns string.

22.1.28 stringtools.to_snake_case

`stringtools.to_snake_case(string)`

Changes *string* to snake case.

Changes words to snake case:

```
>>> stringtools.to_snake_case('scale degrees 4 and 5')
'scale_degrees_4_and_5'
```

Changes snake case to snake case:

```
>>> stringtools.to_snake_case('scale_degrees_4_and_5')
'scale_degrees_4_and_5'
```

Changes dash case to snake case:

```
>>> stringtools.to_snake_case('scale-degrees-4-and-5')
'scale_degrees_4_and_5'
```

Changes snake case to snake case:

```
>>> stringtools.to_snake_case('ScaleDegrees4And5')
'scale_degrees_4_and_5'
```

Returns string.

22.1.29 `stringtools.to_space_delimited_lowercase`

`stringtools.to_space_delimited_lowercase` (*string*)

Changes *string* to space-delimited lowercase.

Changes upper camel case *string* to space-delimited lowercase:

```
>>> stringtools.to_space_delimited_lowercase('LogicalTie')
'logical tie'
```

Changes underscore-delimited *string* to space-delimited lowercase:

```
>>> stringtools.to_space_delimited_lowercase('logical_tie')
'logical tie'
```

Returns space-delimited string unchanged:

```
>>> stringtools.to_space_delimited_lowercase('logical tie')
'logical tie'
```

Returns empty *string* unchanged:

```
>>> stringtools.to_space_delimited_lowercase('')
''
```

Returns string.

22.1.30 `stringtools.to_upper_camel_case`

`stringtools.to_upper_camel_case` (*string*)

Changes *string* to upper camel case.

Changes words to upper camel case:

```
>>> stringtools.to_upper_camel_case('scale degrees 4 and 5')
'ScaleDegrees4And5'
```

Changes snake case to upper camel case:

```
>>> stringtools.to_upper_camel_case('scale_degrees_4_and_5')
'ScaleDegrees4And5'
```

Changes dash case to upper camel case:

```
>>> stringtools.to_upper_camel_case('scale-degrees-4-and-5')
'ScaleDegrees4And5'
```

Changes upper camel case to upper camel case:

```
>>> stringtools.to_upper_camel_case('ScaleDegrees4And5')
'ScaleDegrees4And5'
```

Returns string.

22.1.31 `stringtools.upper_camel_case_to_snake_case`

`stringtools.upper_camel_case_to_snake_case` (*string*)

Changes upper camel case *string* to snake-case.

```
>>> string = 'KeySignature'
```

```
>>> stringtools.upper_camel_case_to_snake_case(string)
'key_signature'
```

Returns string.

22.1.32 stringtools.upper_camel_case_to_space_delimited_lowercase

`stringtools.upper_camel_case_to_space_delimited_lowercase` (*string*)

Changes upper camel case *string* to space-delimited lowercase.

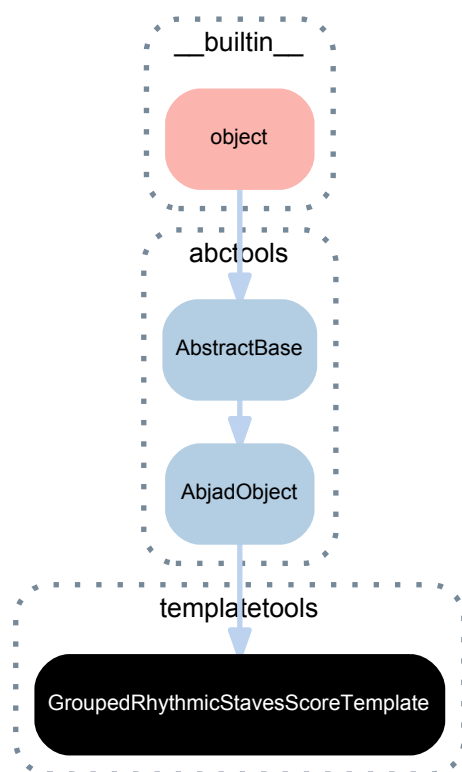
```
>>> string = 'KeySignature'
```

```
>>> stringtools.upper_camel_case_to_space_delimited_lowercase(string)
'key signature'
```

Returns string.

23.1 Concrete classes

23.1.1 `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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

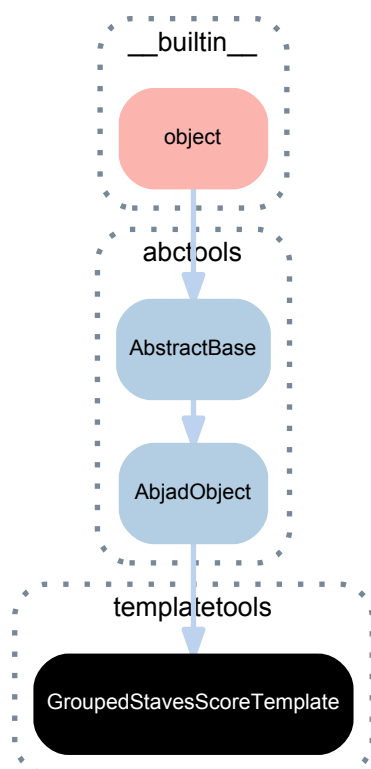
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

23.1.2 `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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

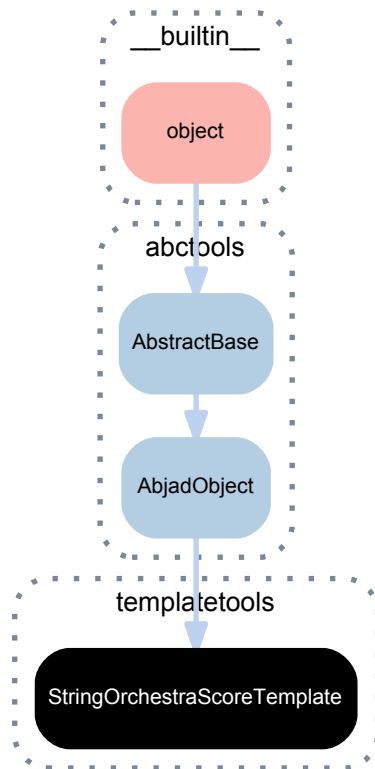
`(AbjadObject).__format__(format_specification='')`
Formats Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

23.1.3 `templatetools.StringOrchestraScoreTemplate`



class `templatetools.StringOrchestraScoreTemplate` (*violin_count=6, viola_count=4, cello_count=3, contra-*
bass_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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

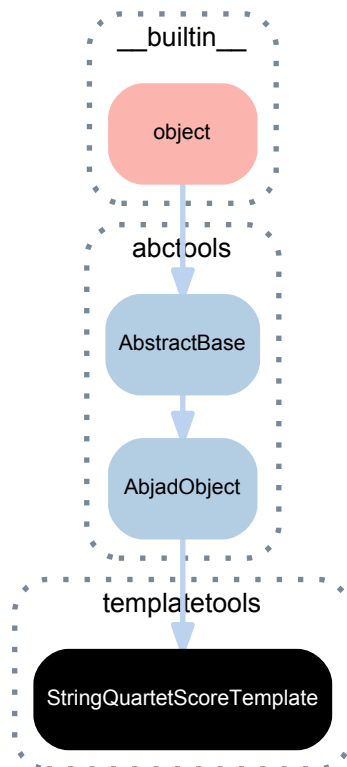
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

23.1.4 `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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

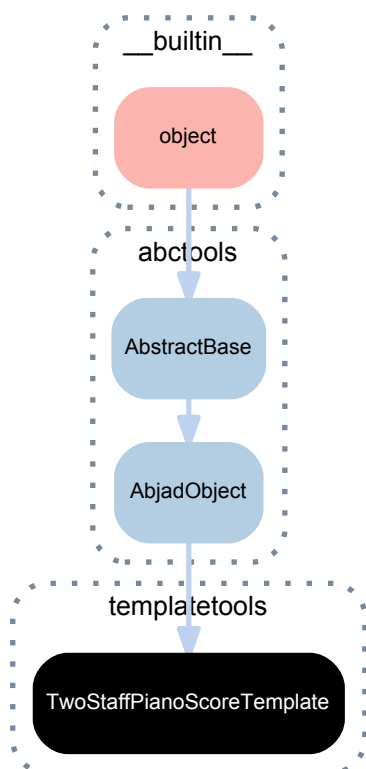
(AbjadObject).**__ne__**(*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

Returns string.

23.1.5 `templatetools.TwoStaffPianoScoreTemplate`



class `templatetools.TwoStaffPianoScoreTemplate`
Two-staff piano score template.

```
>>> template = templatetools.TwoStaffPianoScoreTemplate()
>>> score = template()
```

```
>>> score
<Score-"Two-Staff Piano Score"<<1>>>
```

```
>>> print(format(score))
\context Score = "Two-Staff Piano Score" <<
  \context PianoStaff = "Piano Staff" <<
    \set PianoStaff.instrumentName = \markup { Piano }
    \set PianoStaff.shortInstrumentName = \markup { Pf. }
    \context Staff = "RH Staff" {
      \clef "treble"
      \context Voice = "RH Voice" {
        }
```

```

    }
    \context Staff = "LH Staff" {
        \clef "bass"
        \context Voice = "LH Voice" {
            }
        }
    }
>>
>>

```

Returns score template.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

Returns boolean.

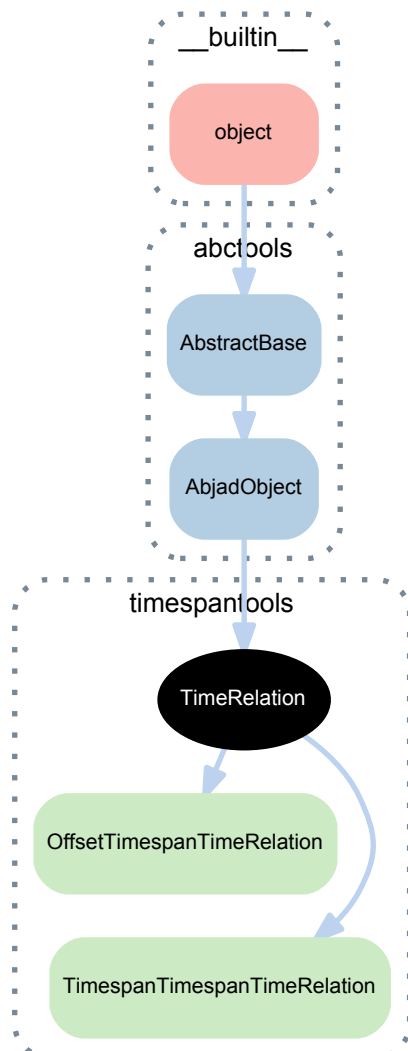
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

24.1 Abstract classes

24.1.1 timespantools.TimeRelation



```
class timespantools.TimeRelation (inequality=None)
    A time relation.

    Time relations are immutable.
```

Bases

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

Read-only properties

`TimeRelation.inequality`

Time relation inequality.

Return inequality.

`TimeRelation.is_fully_loaded`

Is true when both time relation terms are not none. Otherwise false:

Returns boolean.

`TimeRelation.is_fully_unloaded`

Is true when both time relation terms are none. Otherwise false:

Returns boolean.

Special methods

`TimeRelation.__call__()`

Evaluates time relation.

Returns boolean.

`TimeRelation.__eq__(expr)`

Is true when *expr* is a equal-valued time relation. Otherwise false.

Returns boolean.

`TimeRelation.__format__(format_specification='')`

Formats time relation.

Returns string.

`TimeRelation.__hash__()`

Hashes time relation.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

Returns boolean.

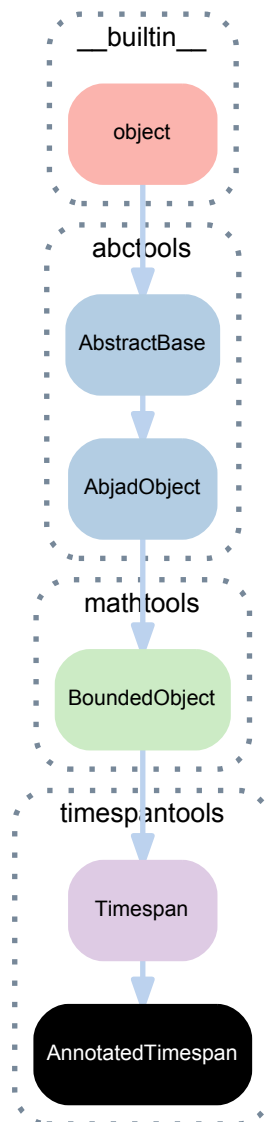
`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

24.2 Concrete classes

24.2.1 timespantools.AnnotatedTimespan



class `timespantools.AnnotatedTimespan` (*start_offset=NegativeInfinity*, *stop_offset=Infinity*, *annotation=None*)

An annotated timespan.

```

>>> annotated_timespan = timespantools.AnnotatedTimespan(
...     annotation=['a', 'b', 'c'],
...     start_offset=Offset(1, 4),
...     stop_offset=Offset(7, 8),
... )
>>> print(format(annotated_timespan))
timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(1, 4),
    stop_offset=durationtools.Offset(7, 8),
    annotation=['a', 'b', 'c'],
)
  
```

Annotated timespans maintain their annotations duration mutation:

```

>>> left, right = annotated_timespan.split_at_offset(Offset(1, 2))
>>> left.annotation.append('foo')
>>> print(format(right))
  
```

```
timespantools.AnnotatedTimespan(  
    start_offset=durationtools.Offset(1, 2),  
    stop_offset=durationtools.Offset(7, 8),  
    annotation=['a', 'b', 'c', 'foo'],  
)
```

Bases

- `timespantools.Timespan`
- `mathtools.BoundedObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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`

Is 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.

Read/write properties

`AnnotatedTimespan.annotation`

Gets and sets annotated timespan annotation.

Gets annotation:

```
>>> annotated_timespan.annotation
['a', 'b', 'c', 'foo']
```

Sets annotation:

```
>>> annotated_timespan.annotation = 'baz'
```

Returns arbitrary object.

Methods

(Timespan).**contains_timespan_improperly**(timespan)

Is 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)

Is 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)

Is 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*)

Is 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*)

Is 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)`
Is 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)`
Is 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)`
Is 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)`
Is 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*)
Is 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*)
Is true when timespan overlaps start of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_start_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_start_of_timespan(timespan_2)
False
>>> timespan_1.overlaps_start_of_timespan(timespan_3)
True
>>> timespan_1.overlaps_start_of_timespan(timespan_4)
True
```

Returns boolean.

(Timespan).**overlaps_stop_of_timespan**(*timespan*)
Is 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).**split_at_offsets** (*offsets*)

Split into one or more parts when *offsets* happens during timespan:

```
>>> timespan = timespantools.Timespan(0, 10)
```

```
>>> shards = timespan.split_at_offsets((1, 3, 7))
>>> for shard in shards:
...     shard
...
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(1, 1))
Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 1))
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(7, 1))
Timespan(start_offset=Offset(7, 1), stop_offset=Offset(10, 1))
```

Otherwise return a tuple containing a copy of timespan:

```
>>> timespan.split_at_offsets((-100,))
(Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1)),)
```

Returns one or more newly constructed timespans.

(Timespan).**starts_after_offset** (*offset*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)
Is 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*)
Is 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*)
Is true when timespan stops after *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.stops_after_offset(Offset(-5))
True
>>> timespan_1.stops_after_offset(Offset(0))
False
>>> timespan_1.stops_after_offset(Offset(5))
False
```

Returns boolean.

(Timespan).**stops_after_timespan_starts**(*timespan*)
Is 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)`

Is 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)`

Is 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)`

Is 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)`

Is 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)`

Is 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)`
Is 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)`
Is 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)`
Is 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)`
Is 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*)

Is 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*)

Is 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*)
Is 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)`

Is 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)`

Is 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) .__hash__()`

Hashes timespan.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(Timespan) .__le__(expr)`

Is 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)`

Is true when *expr* start offset is less than timespan start offset.

```
>>> timespan_1 < timespan_2
True
```

Otherwise false:

```
>>> timespan_2 < timespan_3
False
```

Returns boolean.

`(Timespan) .__ne__(timespan)`

Is 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),
    ),
  ]
)

```

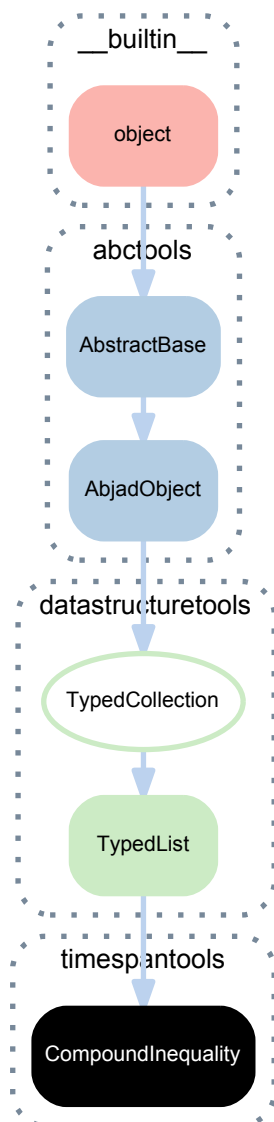
```

>>> 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.

24.2.2 timespantools.CompoundInequality



class `timespantools.CompoundInequality` (*items=None, logical_operator='and'*)
 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',
        ),
    ],
    logical_operator='or',
)
```



```

        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',
)

```

Bases

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

Read-only properties

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

`(TypedCollection).items`
Gets collection items.

`CompoundInequality.logical_operator`
Compound inequality logical operator.

Read/write properties

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

Methods

`(TypedList).append(item)`
Changes *item* 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(item)`
Changes *item* to item and returns count.

```
>>> integer_collection = datastructuretools.TypedList(  
...     items=[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** (*items*)

Changes *items* 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** (*item*)

Changes *item* to item and returns index.

```
>>> pitch_collection = datastructuretools.TypedList(  
...     items=('cquf', 'as', 'b', 'dss'),  
...     item_class=NamedPitch)  
>>> pitch_collection.index("as")  
1
```

Returns index.

(`TypedList`) **.insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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__** (*item*)

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

(`TypedList`) . **__delitem__** (*i*)

Aliases `list.__delitem__()`.

Returns none.

(`TypedCollection`) . **__eq__** (*expr*)

Is 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__()`.

Returns item.

(`TypedCollection`) . **__hash__** ()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(`TypedList`) . **__iadd__** (*expr*)

Changes items 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(
      name='ppp',
    ),
    indicatortools.Dynamic(
      name='p',
    ),
    indicatortools.Dynamic(
      name='mp',
    ),
    indicatortools.Dynamic(
      name='mf',
    ),
    indicatortools.Dynamic(
      name='fff',
    ),
  ],
  item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection).**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection).**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__**().

Returns generator.

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

Changes items 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'"),
  ]
)
```

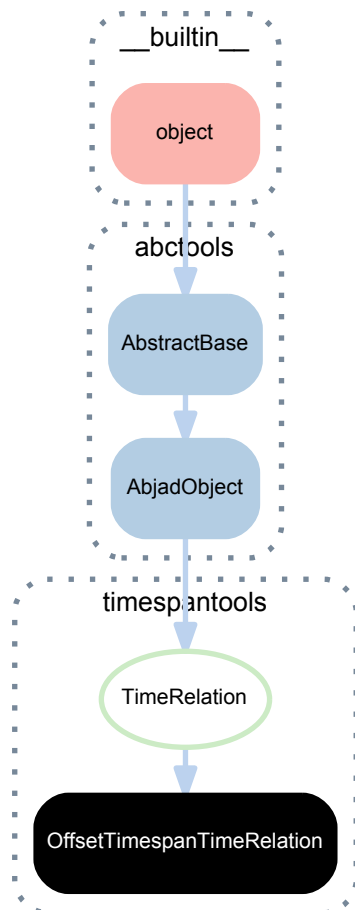
```

pitchtools.NamedPitch("g' "),
pitchtools.NamedPitch("a' "),
pitchtools.NamedPitch("b' "),
pitchtools.NamedPitch("c' "),
],
item_class=pitchtools.NamedPitch,
)

```

Returns none.

24.2.3 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(TimeRelation).inequality`

Time relation inequality.

Return inequality.

`OffsetTimespanTimeRelation.is_fully_loaded`

Is true when *timespan* and *offset* are both not none. Otherwise false:

```
>>> time_relation.is_fully_loaded
True
```

Returns boolean.

`OffsetTimespanTimeRelation.is_fully_unloaded`

Is 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)`
Is 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.

`OffsetTimespanTimeRelation.__hash__()`
Hashes time relation.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

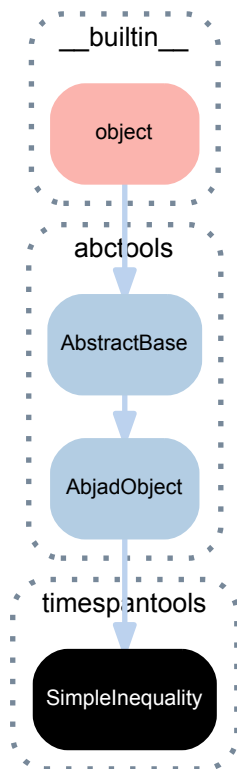
`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

Returns string.

24.2.4 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`
- `abctools.AbjadObject.AbstractBase`
- `__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')
```

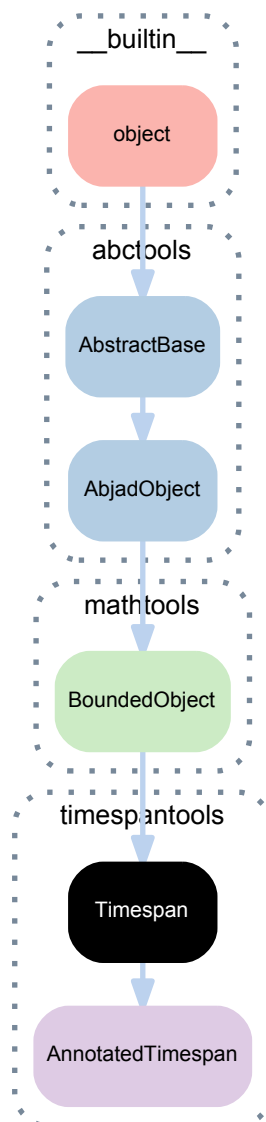
Returns string.

`(AbjadObject).__hash__()`
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 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.

24.2.5 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`
- `abctools.AbjadObject.AbstractBase`
- `__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
Is 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*)
Is 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*)
Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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.split_at_offsets` (*offsets*)

Split into one or more parts when *offsets* happens during timespan:

```
>>> timespan = timespantools.Timespan(0, 10)

>>> shards = timespan.split_at_offsets((1, 3, 7))
>>> for shard in shards:
...     shard
...
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(1, 1))
Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 1))
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(7, 1))
Timespan(start_offset=Offset(7, 1), stop_offset=Offset(10, 1))
```

Otherwise return a tuple containing a copy of timespan:

```
>>> timespan.split_at_offsets((-100,))
(Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1)),)
```

Returns one or more newly constructed timespans.

`Timespan.starts_after_offset` (*offset*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is 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*)

Is true when timespan stops before *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)

>>> timespan_1.stops_before_timespan_stops(timespan_1)
False
>>> timespan_1.stops_before_timespan_stops(timespan_2)
True
>>> timespan_2.stops_before_timespan_stops(timespan_1)
False
```

```
False
>>> timespan_2.stops_before_timespan_stops(timespan_2)
False
```

Returns boolean.

Timespan.stops_during_timespan (*timespan*)

Is 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*)

Is 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*)

Is 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*)
 Is 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)`

Is 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)`

Is 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)`

Is 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.__hash__()`

Hashes timespan.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`Timespan.__le__(expr)`

Is 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)`

Is true when *expr* start offset is less than timespan start offset.

```
>>> timespan_1 < timespan_2
True
```

Otherwise false:

```
>>> timespan_2 < timespan_3
False
```

Returns boolean.

`Timespan.__ne__(timespan)`

Is 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),
        ),
    ]
)

```

```

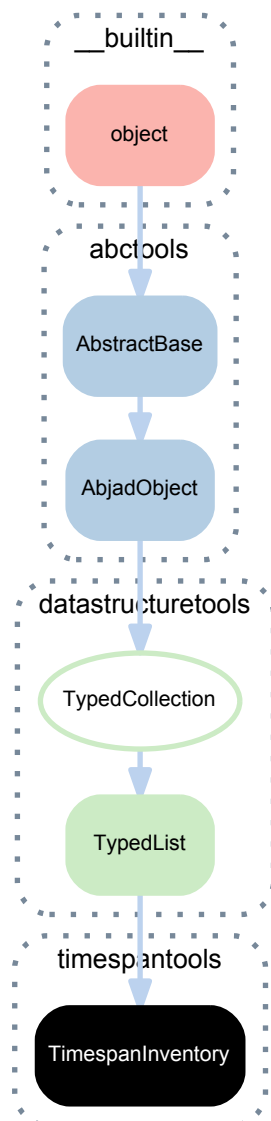
>>> new_timespan = timespan_2 ^ timespan_4
>>> print(format(new_timespan))
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(5, 1),
            stop_offset=durationtools.Offset(10, 1),
        ),
        timespantools.Timespan(
            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.

24.2.6 timespantools.TimespanInventory



class `timespantools.TimespanInventory` (*items=None, item_class=None, keep_sorted=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`
- `abctools.AbjadObject.AbstractBase`

- `__builtin__.object`

Read-only properties

`TimespanInventory.all_are_contiguous`

Is 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
```

Is true when empty:

```
>>> timespan_inventory_3.all_are_contiguous
True
```

Returns boolean.

`TimespanInventory.all_are_nonoverlapping`

Is 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
```

Is true when empty:

```
>>> timespan_inventory_3.all_are_nonoverlapping
True
```

Returns boolean.

`TimespanInventory.all_are_well_formed`

Is 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

Is 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 items into.

(TypedCollection).items

Gets collection items.

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

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

Methods

(TypedList).append(item)
Changes *item* 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.clip_timespan_durations (*minimum=None, maximum=None, anchor=Left*)

Clip durations of timespans.

Example 1:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 1),
...     timespantools.Timespan(0, 10),
... ])
>>> result = timespan_inventory.clip_timespan_durations(
...     minimum=5,
... )
>>> print(format(result))
timespantools.TimespanInventory(
[
```

```

        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(5, 1),
        ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(10, 1),
        ),
    ]
)

```

Example 2:

```

>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 1),
...     timespantools.Timespan(0, 10),
... ])
>>> result = timespan_inventory.clip_timespan_durations(
...     maximum=5,
... )
>>> print(format(result))
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(1, 1),
        ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(5, 1),
        ),
    ]
)

```

Example 3:

```

>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 1),
...     timespantools.Timespan(0, 10),
... ])
>>> result = timespan_inventory.clip_timespan_durations(
...     minimum=3,
...     maximum=7,
... )
>>> print(format(result))
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(3, 1),
        ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(7, 1),
        ),
    ]
)

```

Example 4:

```

>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 1),
...     timespantools.Timespan(0, 10),
... ])
>>> result = timespan_inventory.clip_timespan_durations(
...     minimum=3,
...     maximum=7,
...     anchor=Right,
... )
>>> print(format(result))
timespantools.TimespanInventory(
    [

```

```
        timespantools.Timespan(
            start_offset=durationtools.Offset(-2, 1),
            stop_offset=durationtools.Offset(1, 1),
        ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(3, 1),
            stop_offset=durationtools.Offset(10, 1),
        ),
    ]
)
```

Emit new inventory.

`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.items():
...     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(item)`
 Changes *item* to item and returns count.

```
>>> integer_collection = datastructuretools.TypedList(
...     items=[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().items()):
...     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().items()):
...     offset, count
...
(Offset(0, 1), 1)
(Offset(3, 1), 1)
(Offset(6, 1), 1)
(Offset(10, 1), 1)
(Offset(15, 1), 1)
(Offset(20, 1), 1)
```

Example 3:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(0, 6),
...     timespantools.Timespan(0, 9),
... ])
>>> for offset, count in sorted(
...     timespan_inventory.count_offsets().items()):
...     offset, count
...
(Offset(0, 1), 3)
(Offset(3, 1), 1)
(Offset(6, 1), 1)
```

```
(Offset(9, 1), 1)
```

Returns counter.

`TimespanInventory.explode` (*inventory_count=None*)

Explode timespans into inventories, avoiding overlap, and distributing density as evenly as possible.

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(5, 13),
...     timespantools.Timespan(6, 10),
...     timespantools.Timespan(8, 9),
...     timespantools.Timespan(15, 23),
...     timespantools.Timespan(16, 21),
...     timespantools.Timespan(17, 19),
...     timespantools.Timespan(19, 20),
...     timespantools.Timespan(25, 30),
...     timespantools.Timespan(26, 29),
...     timespantools.Timespan(32, 34),
...     timespantools.Timespan(34, 37),
... ])
```

Example 1. Explode timespans into the optimal number of non-overlapping inventories:

```
>>> for exploded_inventory in timespan_inventory.explode():
...     print(format(exploded_inventory))
...
timespantools.TimespanInventory(
  [
    timespantools.Timespan(
      start_offset=durationtools.Offset(0, 1),
      stop_offset=durationtools.Offset(3, 1),
    ),
    timespantools.Timespan(
      start_offset=durationtools.Offset(5, 1),
      stop_offset=durationtools.Offset(13, 1),
    ),
    timespantools.Timespan(
      start_offset=durationtools.Offset(17, 1),
      stop_offset=durationtools.Offset(19, 1),
    ),
    timespantools.Timespan(
      start_offset=durationtools.Offset(19, 1),
      stop_offset=durationtools.Offset(20, 1),
    ),
    timespantools.Timespan(
      start_offset=durationtools.Offset(34, 1),
      stop_offset=durationtools.Offset(37, 1),
    ),
  ]
)
timespantools.TimespanInventory(
  [
    timespantools.Timespan(
      start_offset=durationtools.Offset(6, 1),
      stop_offset=durationtools.Offset(10, 1),
    ),
    timespantools.Timespan(
      start_offset=durationtools.Offset(16, 1),
      stop_offset=durationtools.Offset(21, 1),
    ),
    timespantools.Timespan(
      start_offset=durationtools.Offset(25, 1),
      stop_offset=durationtools.Offset(30, 1),
    ),
  ]
)
timespantools.TimespanInventory(
  [
    timespantools.Timespan(
      start_offset=durationtools.Offset(8, 1),
      stop_offset=durationtools.Offset(9, 1),

```

```
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(23, 1),
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(26, 1),
        stop_offset=durationtools.Offset(29, 1),
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(32, 1),
        stop_offset=durationtools.Offset(34, 1),
    ),
]
)
```

Example 2. Explode timespans into a less-than-optimal number of overlapping inventories:

```
>>> for exploded_inventory in timespan_inventory.explode(
...     inventory_count=2):
...     print(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** (*items*)

Changes *items* 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*)
Is 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** (*item*)

Changes *item* to item and returns index.

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

Returns index.

(`TypedList`) **.insert** (*i*, *item*)

Changes *item* 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** (*item*)

Changes *item* 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__**(*item*)

Is true when typed collection container *item*. Otherwise false.

Returns boolean.

(TypedList) .**__delitem__**(*i*)

Aliases list.**__delitem__**().

Returns none.

(TypedCollection) .**__eq__**(*expr*)

Is 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__**().

Returns item.

(TypedCollection) .**__hash__**()

Hashes typed collection.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TypedList) .**__iadd__**(*expr*)

Changes items 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(
        name='ppp',
    ),
    indicatortools.Dynamic(
        name='p',
    ),
    indicatortools.Dynamic(
        name='mp',
    ),
    indicatortools.Dynamic(
        name='mf',
    ),
    indicatortools.Dynamic(
        name='fff',
    ),
],
item_class=indicatortools.Dynamic,
)
```

Returns collection.

(TypedCollection) .**__iter__**()

Iterates typed collection.

Returns generator.

(TypedCollection) .**__len__**()

Length of typed collection.

Returns nonnegative integer.

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

Is 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__**().

Returns generator.

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

Changes items 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,
)
```

Returns none.

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),
```

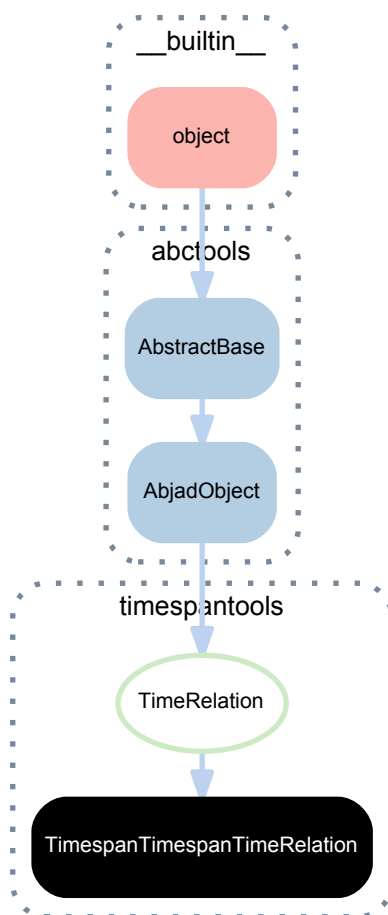
```

    ),
    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.

24.2.7 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

(TimeRelation).**.inequality**

Time relation inequality.

Return inequality.

TimespanTimespanTimeRelation.**.is_fully_loaded**

Is 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**

Is 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)`
Is 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.

TimespanTimespanTimeRelation.**__hash__**()

Hashes time relation.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

24.3 Functions

24.3.1 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.

24.3.2 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.

24.3.3 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.

24.3.4 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.

24.3.5 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.

24.3.6 `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.

24.3.7 `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.

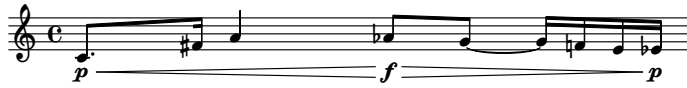
24.3.8 `timespantools.timespan_2_contains_timespan_1_improperly`

`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.

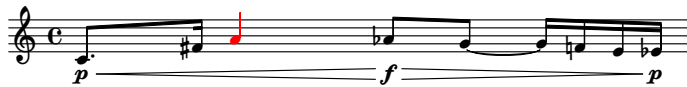
```
>>> relation = timespantools.timespan_2_contains_timespan_1_improperly()  
>>> print(format(relation))  
timespantools.TimespanTimespanTimeRelation(  
    inequality=timespantools.CompoundInequality(  
        [  
            timespantools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),  
            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.

24.3.9 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.TimeSpanTimeRelation(
    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.

24.3.10 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.TimeSpanTimeRelation(
    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.

24.3.11 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.

24.3.12 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.

24.3.13 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.

24.3.14 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.

24.3.15 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.

24.3.16 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.

24.3.17 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.

24.3.18 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.

24.3.19 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.

24.3.20 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.

24.3.21 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.

24.3.22 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.

24.3.23 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.

24.3.24 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.

24.3.25 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.

24.3.26 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.

24.3.27 `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.

24.3.28 `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.

24.3.29 `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.

24.3.30 `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.

24.3.31 `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.

24.3.32 `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.

24.3.33 `timespantools.timespan_2_trisects_timespan_1`

`timespantools.timespan_2_trisects_timespan_1` (*timespan_1=None, timespan_2=None, hold=False*)

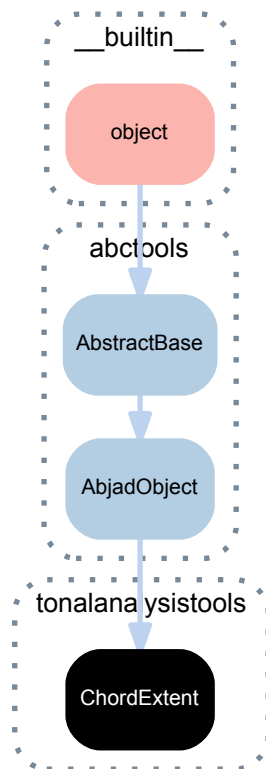
Makes time relation indicating that *timespan_2* trisects *timespan_1*.

```
>>> 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.

25.1 Concrete classes

25.1.1 tonalanalysistools.ChordExtent



class `tonalanalysistools.ChordExtent` (*number=5*)
A chord extent, such as triad, seventh chord, ninth chord, etc.
Value object that can not be changed after instantiation.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`ChordExtent.__hash__()`

Hashes chord extent.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`ChordExtent.__ne__(arg)`

Is true when chord extent does not equal *arg*. Otherwise false.

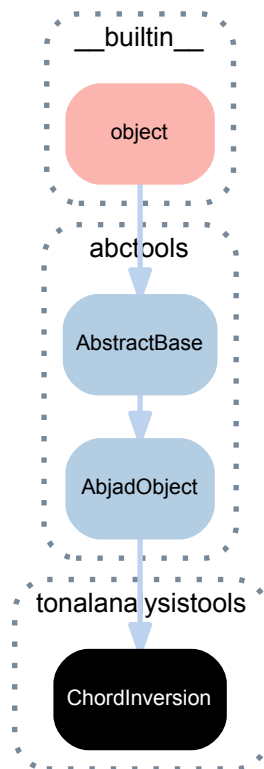
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

25.1.2 tonalanalysistools.ChordInversion



class `tonalanalysistools.ChordInversion` (*number=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`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

Is 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 Abjad object.

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

Returns string.

`ChordInversion.__hash__` ()

Hashes chord inversion.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`ChordInversion.__ne__` (*arg*)

Is true when chord inversion does not equal *arg*. Otherwise false.

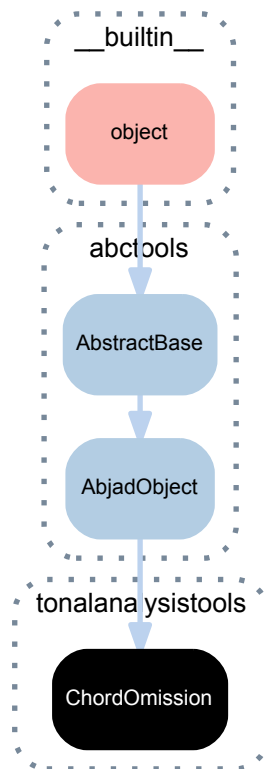
Returns boolean.

`(AbjadObject).__repr__` ()

Gets interpreter representation of Abjad object.

Returns string.

25.1.3 tonalanalysistools.ChordOmission



class `tonalanalysistools.ChordOmission`

A chord omission.

Value object that can not be changed after instantiation.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

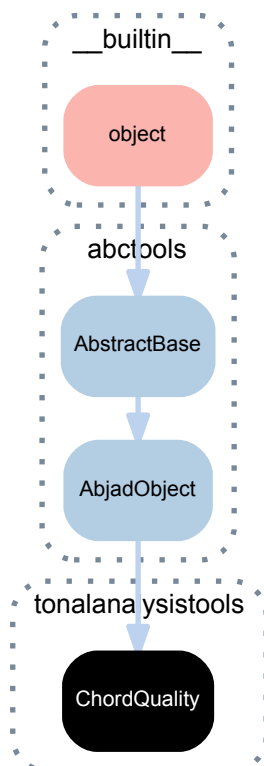
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

25.1.4 tonalanalysistools.ChordQuality



class tonalanalysistools.**ChordQuality** (*quality_string*=*'major'*)

A chord quality, such as major, minor, dominant, diminished and so on.

Value object that can not be changed after instantiation.

Bases

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

Read-only properties

`ChordQuality.is_uppercase`

Is 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)`

Is 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 Abjad object.

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

Returns string.

`ChordQuality.__hash__()`

Hashes chord quality.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`ChordQuality.__ne__(arg)`

Is true when chord quality does not equal *arg*. Otherwise false.

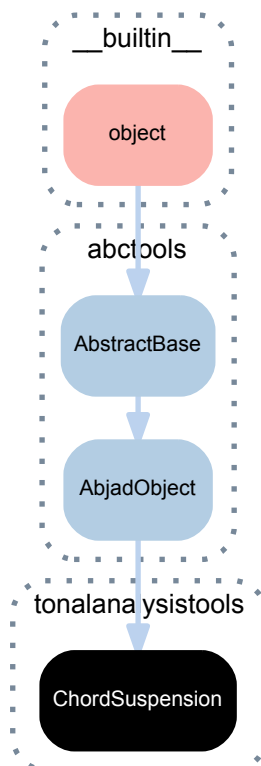
Returns boolean.

`ChordQuality.__repr__()`

Gets interpreter representation of chord quality.

Returns string.

25.1.5 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`
- `abctools.AbjadObject.AbstractBase`
- `__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`
Is 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)`

Is 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 Abjad object.

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

Returns string.

`ChordSuspension.__hash__()`

Hashes chord suspension.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`ChordSuspension.__ne__(arg)`

Is true when *arg* does not equal chord suspension. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

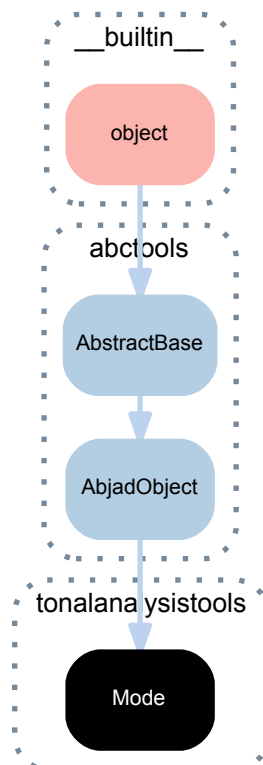
Returns string.

`ChordSuspension.__str__()`

String representation of chord suspension.

Returns string.

25.1.6 tonalanalysistools.Mode



class `tonalanalysistools.Mode` (*mode_name*='dorian')

A diatonic mode.

Can be extended for nondiatonic mode.

Modes with different ascending and descending forms not yet implemented.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

Is true when *arg* is a mode with mode name equal to that of this mode. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`Mode.__hash__` ()

Hashes mode.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`Mode.__len__` ()

Length of mode.

Returns nonnegative integer.

`Mode.__ne__` (*arg*)

Is true when *arg* does not equal mode. Otherwise false.

Returns boolean.

(`AbjadObject`). `__repr__` ()

Gets interpreter representation of Abjad object.

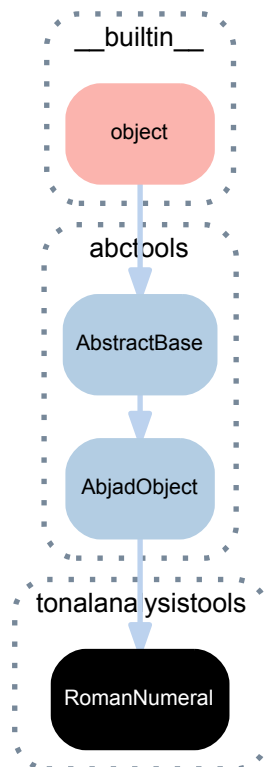
Returns string.

`Mode.__str__` ()

String representation of mode.

Returns string.

25.1.7 tonalanalysisistools.RomanNumeral



class `tonalanalysisistools.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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`RomanNumeral.__hash__()`

Hashes roman numeral.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`RomanNumeral.__ne__(arg)`

Is true when roman numeral does not equal *arg*. Otherwise false.

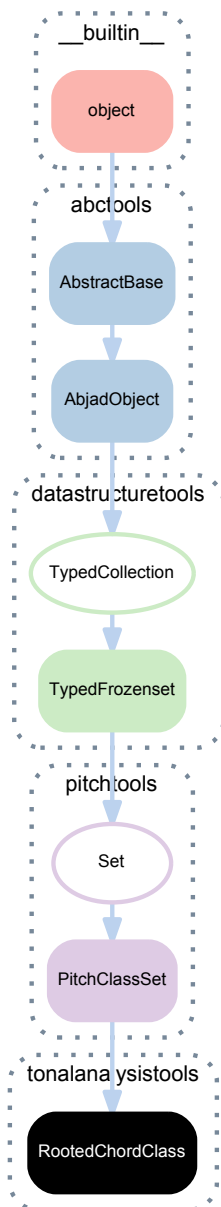
Returns boolean.

`RomanNumeral.__repr__()`

Gets interpreter representation of *arg*.

Returns string.

25.1.8 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`
- `abctools.AbjadObject.AbstractBase`
- `__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(number=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 items into.

`(TypedCollection).items`

Gets collection items.

`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.

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)`
Is 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)`

Is 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)`

Is true when typed frozen set shares no elements with *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenset).issubset(expr)`

Is true when typed frozen set is a subset of *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenset).issuperset(expr)`

Is 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)`
 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__(item)`
 Is true when typed collection container *item*. Otherwise false.

Returns boolean.

`RootedChordClass.__eq__(arg)`
 Is 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)`

Is true when typed frozen set is greater than or equal to *expr*. Otherwise false.

Returns boolean.

`(TypedFrozenSet) .__gt__(expr)`

Is true when typed frozen set is greater than *expr*. Otherwise false.

Returns boolean.

`RootedChordClass .__hash__()`

Hashes rooted chord-class.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(TypedCollection) .__iter__()`

Iterates typed collection.

Returns generator.

`(TypedFrozenSet) .__le__(expr)`

Is 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)`

Is true when typed frozen set is less than *expr*. Otherwise false.

Returns boolean.

`RootedChordClass .__ne__(arg)`

Is 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.

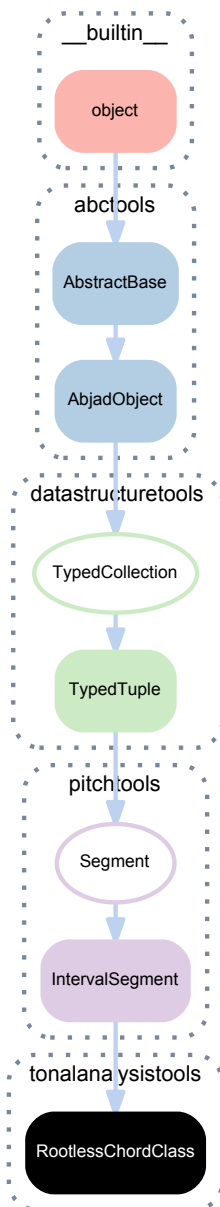
Returns new typed frozen set.

`(TypedFrozenSet) .__xor__(expr)`

Logical XOR of typed frozen set and *expr*.

Returns new typed frozen set.

25.1.9 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 items into.

`(TypedCollection)`.**`items`**

Gets collection items.

`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.

Methods

`(TypedTuple)`.**count** (*item*)

Changes *item* to item.

Returns count in collection.

`(TypedTuple)`.**index** (*item*)

Changes *item* 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*)

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__** (*item*)

Change *item* to item and return true if item exists in collection.

Returns none.

(TypedCollection) .**__eq__** (*expr*)

Is 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.

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

Multiplies typed tuple by *expr*.

Returns new typed tuple.

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

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

Returns boolean.

(TypedTuple) .**__radd__** (*expr*)

Right-adds *expr* to typed tuple.

RootlessChordClass .**__repr__** ()

Gets interpreter representation of rootless chord-class.

Returns string.

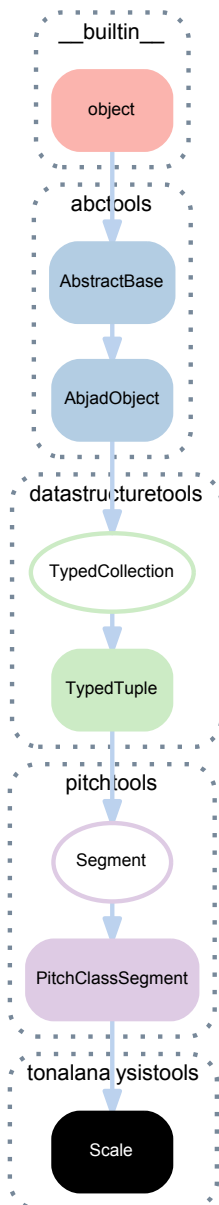
(TypedTuple) .**__rmul__** (*expr*)

Multiplies *expr* by typed tuple.

Returns new typed tuple.

(Segment).**__str__**()
 String representation of segment.
 Returns string.

25.1.10 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 (
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     )
>>> pitch_class_segment.has_duplicates
True
```

```
>>> pitch_class_segment = pitchtools.PitchClassSegment (
...     items="c d e f g a b",
...     )
>>> pitch_class_segment.has_duplicates
False
```

Returns boolean.

`(TypedCollection).item_class`

Item class to coerce items into.

`(TypedCollection).items`

Gets collection items.

`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.

Methods

`(PitchClassSegment)`.**alpha**()
 Morris alpha transform of pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(
...     items=[-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**(*item*)
 Changes *item* 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**(*item*)
 Changes *item* to item.
 Returns index in collection.

`(PitchClassSegment)`.**invert**()
 Invert pitch-class segment:

```
>>> pitch_class_segment = pitchtools.PitchClassSegment(
...     items=[-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 = tonalanalysistools.Scale('c', 'major')
>>> notes = scale.make_notes(8)
>>> staff = Staff(notes)
```

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



Allow nonassignable *written_duration*:

```
>>> notes = scale.make_notes(4, Duration(5, 16))
>>> staff = Staff(notes)
>>> time_signature = 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(
...     items=[-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(
...     items=[-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*, transpose=False)**

Rotate pitch-class segment:

```
>>> pitchtools.PitchClassSegment(
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     ).rotate(1)
PitchClassSegment([7, 10, 10.5, 6, 7, 10.5])
```

```
>>> pitchtools.PitchClassSegment(
...     items=['c', 'ef', 'bqs', 'd'],
...     ).rotate(-2)
PitchClassSegment(['bqs', 'd', 'c', 'ef'])
```

If *transpose* is true, transpose the rotated segment to begin at the same pitch class as this segment:

```
>>> pitchtools.PitchClassSegment (
...     items=['c', 'b', 'd']
...     ).rotate(1, transpose=True)
PitchClassSegment(['c', 'bf', 'a'])
```

Returns new pitch-class segment.

Scale.**scale_degree_to_named_pitch_class** (*args)

Changes scale degree to named pitch-class.

```
>>> scale = tonalanalysistools.Scale('c', 'major')
>>> scale_degree = tonalanalysistools.ScaleDegree('flat', 5)
>>> scale.scale_degree_to_named_pitch_class(scale_degree)
NamedPitchClass('gf')
```

```
>>> scale_degree = tonalanalysistools.ScaleDegree('flat', 9)
>>> scale.scale_degree_to_named_pitch_class(scale_degree)
NamedPitchClass('df')
```

Returns named pitch-class.

(PitchClassSegment).**transpose** (expr)

Transpose pitch-class segment:

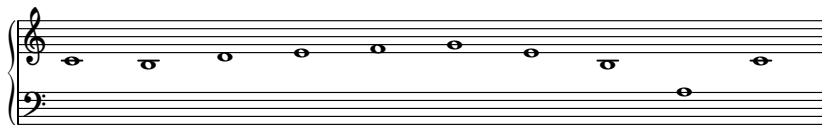
```
>>> pitchtools.PitchClassSegment (
...     items=[-2, -1.5, 6, 7, -1.5, 7],
...     ).transpose(10)
PitchClassSegment([8, 8.5, 4, 5, 8.5, 5])
```

Returns new pitch-class segment.

(PitchClassSegment).**voice_horizontally** (initial_octave=4)

Voices pitch-class segment as pitch segment, with each pitch as close in distance to the previous pitch as possible.

```
>>> pitch_classes = pitchtools.PitchClassSegment (
...     "c b d e f g e b a c")
>>> pitch_segment = pitch_classes.voice_horizontally()
>>> show(pitch_segment)
```



Returns pitch segment.

Scale.**voice_scale_degrees_in_open_position** (scale_degrees)

Voice *scale_degrees* in open position:

```
>>> scale = tonalanalysistools.Scale('c', 'major')
>>> scale_degrees = [1, 3, ('flat', 5), 7, ('sharp', 9)]
>>> pitches = scale.voice_scale_degrees_in_open_position(
...     scale_degrees)
>>> pitches
PitchSegment(["c'", "e'", "gf'", "b'", "ds'"])
```

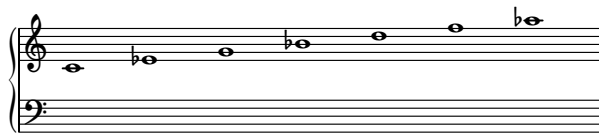
Return pitch segment.

(PitchClassSegment).**voice_vertically** (initial_octave=4)

Voices pitch-class segment as pitch segment, with each pitch always higher than the previous.

```
>>> scale_degree_numbers = [1, 3, 5, 7, 9, 11, 13]
>>> scale = tonalanalysistools.Scale('c', 'minor')
>>> pitch_classes = pitchtools.PitchClassSegment((
...     scale.scale_degree_to_named_pitch_class(x)
...     for x in scale_degree_numbers))
>>> pitch_segment = pitch_classes.voice_vertically()
>>> pitch_segment
```

```
PitchSegment(["c'", "ef'", "g'", "bf'", "d'", "f'", "af'"])
>>> show(pitch_segment)
```



Returns pitch 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__(item)`

Change *item* to item and return true if item exists in collection.

Returns none.

`(TypedCollection).__eq__(expr)`

Is 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.

(TypedTuple) .**__mul__**(*expr*)
Multiplies typed tuple by *expr*.

Returns new typed tuple.

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

Returns boolean.

(TypedTuple) .**__radd__**(*expr*)
Right-adds *expr* to typed tuple.

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

Returns string.

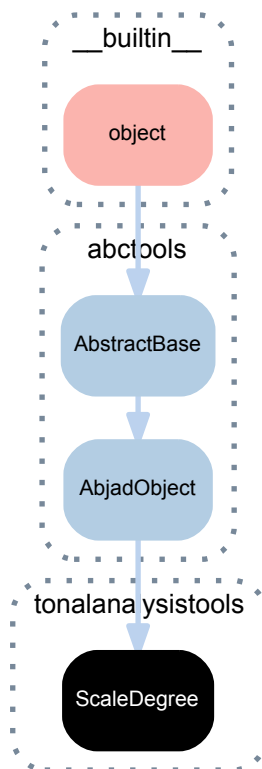
(TypedTuple) .**__rmul__**(*expr*)
Multiplies *expr* by typed tuple.

Returns new typed tuple.

(Segment) .**__str__**()
String representation of segment.

Returns string.

25.1.11 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is 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 Abjad object.

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

Returns string.

`ScaleDegree.__hash__()`

Hashes scale degree.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`ScaleDegree.__ne__(arg)`

Is 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.

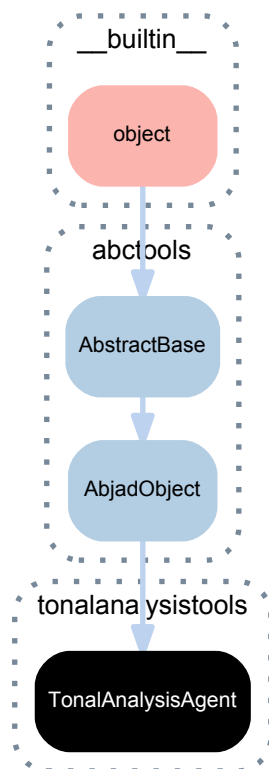
`ScaleDegree.__str__()`

String representation of scale degree.

```
>>> str(scale_degree)
'#4'
```

Returns string.

25.1.12 tonalanalysistools.TonalAnalysisAgent

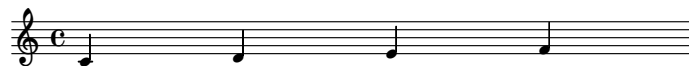


class `tonalanalysistools.TonalAnalysisAgent` (*client=None*)

A tonal analysis interface.

Example 1. Interface to conjunct selection:

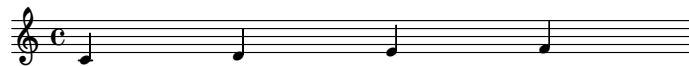
```
>>> staff = Staff("c'4 d' e' f'")
>>> show(staff)
```



```
>>> 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`
- `abctools.AbjadObject.AbstractBase`
- `__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()`
Is 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()`
Is 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` ()
Is 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` ()
Is 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` ()
Is 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` ()
Is 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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

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.

25.2 Functions

25.2.1 tonalanalysistools.select

tonalanalysistools.**select**(*expr*)

Select *expr* for tonal analysis.

Returns tonal analysis selection.

26.1 Functions

26.1.1 `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.

26.1.2 `topleveltools.detach`

`topleveltools.detach` (*prototype*, *component_expression*)
Detaches from *component_expression* all items matching *prototype*.

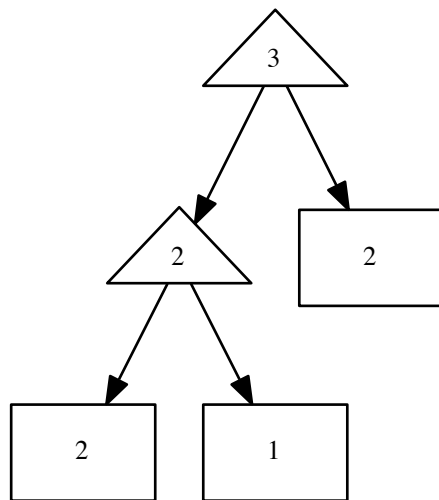
Returns tuple of zero or more detached items.

26.1.3 `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.

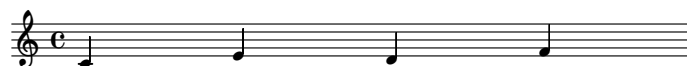
26.1.4 topleveltools.inspect

`topleveltools.inspect_(expr)`
 Inspects *expr*.
 Factory function.
 Returns inspect agent.

26.1.5 topleveltools.iterate

`topleveltools.iterate(expr)`
 iterates *expr*.

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



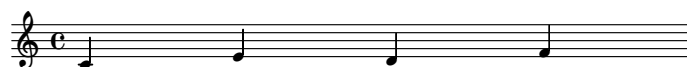
```
>>> notes = staff[-2:]
>>> iterate(notes)
IterationAgent(client=SliceSelection(Note("d'4"), Note("f'4")))
```

Returns score iteration agent.

26.1.6 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.

26.1.7 topleveltools.new

`topleveltools.new` (*expr*, ***kwargs*)

Makes new *expr* with optional new *kwargs*.

Returns new object with the same type as *expr*.

26.1.8 topleveltools.override

`topleveltools.override` (*expr*)

Overrides *expr*.

Returns LilyPond grob manager.

26.1.9 topleveltools.parse

`topleveltools.parse` (*arg*, *language='english'*)

Parses *arg* as LilyPond string.

```
>>> parse("{c'4 d'4 e'4 f'4}")
Container("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')
Container("c'8 df'8 e'8 fs'8")
```

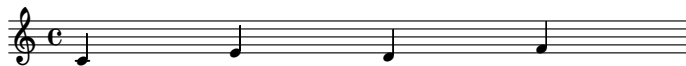
Returns Abjad expression.

26.1.10 topleveltools.persist

`topleveltools.persist` (*expr*)

Persists *expr*.

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



```
>>> persist(staff)
PersistenceAgent(client=Staff("c'4 e'4 d'4 f'4"))
```

Returns score mutation agent.

26.1.11 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 `file_name.mid` under Windows.

Abjad outputs MIDI files of the format `file_name.midi` under other operating systems.

Returns none.

26.1.12 `topleveltools.select`

`topleveltools.select` (*expr=None, contiguous=False*)

Selects *expr*.

Returns selection.

26.1.13 `topleveltools.set`

`topleveltools.set_` (*expr*)

Applies LilyPond context setting to *expr*.

Returns LilyPond context setting manager.

26.1.14 `topleveltools.show`

`topleveltools.show` (*expr, return_timing=False, **kwargs*)

Shows *expr*.

Shows a note:

```
>>> note = Note("c'4")
>>> show(note)
```



Abjad writes LilyPond input files to the `~/ .abjad/output/` directory by default.

You may change this by setting the `abjad_output_directory` variable in the `Abjad config.py` file.

Returns none when *return_timing* is false.

Returns pair of *abjad_formatting_time* and *lilypond_rendering_time* when *return_timing* is true.

Part II

Demos and example packages

27.1 Functions

27.1.1 `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.

27.1.2 `desordre.make_desordre_lilypond_file`

`desordre.make_desordre_lilypond_file` ()

Makes Désordre LilyPond file.

27.1.3 `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.

27.1.4 `desordre.make_desordre_pitches`

`desordre.make_desordre_pitches` ()

Makes Désordre pitches.

27.1.5 `desordre.make_desordre_score`

`desordre.make_desordre_score` (*pitches*)

Returns a complete piano staff with Ligeti music.

27.1.6 `desordre.make_desordre_staff`

`desordre.make_desordre_staff` (*pitches*)

Makes Désordre staff.

28.1 Functions

28.1.1 `ferneyhough.configure_lilypond_file`

`ferneyhough.configure_lilypond_file` (*lilypond_file*)
Configures LilyPond file.

28.1.2 `ferneyhough.configure_score`

`ferneyhough.configure_score` (*score*)
Configured score.

28.1.3 `ferneyhough.make_lilypond_file`

`ferneyhough.make_lilypond_file` (*tuple_duration*, *row_count*, *column_count*)
Makes LilyPond file.

28.1.4 `ferneyhough.make_nested_tuplet`

`ferneyhough.make_nested_tuplet` (*tuple_duration*, *outer_tuplet_proportions*, *inner_tuplet_subdivision_count*)
Makes nested tuplet.

28.1.5 `ferneyhough.make_row_of_nested_tuplets`

`ferneyhough.make_row_of_nested_tuplets` (*tuple_duration*, *outer_tuplet_proportions*, *column_count*)
Makes row of nested tuplets.

28.1.6 `ferneyhough.make_rows_of_nested_tuplets`

`ferneyhough.make_rows_of_nested_tuplets` (*tuple_duration*, *row_count*, *column_count*)
Makes rows of nested tuplets.

28.1.7 `ferneyhough.make_score`

`ferneyhough.make_score` (*tuple_duration*, *row_count*, *column_count*)
Makes score.

29.1 Functions

29.1.1 `mozart.choose_mozart_measures`

`mozart.choose_mozart_measures()`
Chooses Mozart measures.

29.1.2 `mozart.make_mozart_lilypond_file`

`mozart.make_mozart_lilypond_file()`
Makes Mozart LilyPond file.

29.1.3 `mozart.make_mozart_measure`

`mozart.make_mozart_measure(measure_dict)`
Makes Mozart measure.

29.1.4 `mozart.make_mozart_measure_corpus`

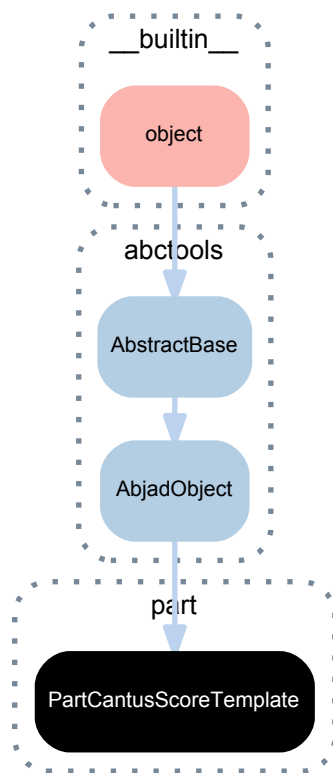
`mozart.make_mozart_measure_corpus()`
Makes Mozart measure corpus.

29.1.5 `mozart.make_mozart_score`

`mozart.make_mozart_score()`
Makes Mozart score.

30.1 Concrete classes

30.1.1 `part.PartCantusScoreTemplate`



class `part.PartCantusScoreTemplate`
Pärt Cantus score template.

Bases

- `abjad.tools.abctools.AbjadObject`
- `abjad.tools.abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject) .**__hash__** ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

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.

30.2 Functions

30.2.1 part.add_bell_music_to_score

`part.add_bell_music_to_score (score)`

Adds bell music to score.

30.2.2 part.add_string_music_to_score

`part.add_string_music_to_score (score)`

Adds string music to score.

30.2.3 part.apply_bowing_marks

`part.apply_bowing_marks (score)`

Applies bowing marks to score.

30.2.4 part.apply_dynamics

`part.apply_dynamics (score)`

Applies dynamics to score.

30.2.5 part.apply_expressive_marks

`part.apply_expressive_marks (score)`

Applies expressive marks to score.

30.2.6 `part.apply_final_bar_lines`

`part.apply_final_bar_lines` (*score*)
Applies final bar lines to score.

30.2.7 `part.apply_page_breaks`

`part.apply_page_breaks` (*score*)
Applies page breaks to score.

30.2.8 `part.apply_rehearsal_marks`

`part.apply_rehearsal_marks` (*score*)
Applies rehearsal marks to score.

30.2.9 `part.configure_lilypond_file`

`part.configure_lilypond_file` (*lilypond_file*)
Configures LilyPond file.

30.2.10 `part.configure_score`

`part.configure_score` (*score*)
Configures score.

30.2.11 `part.create_pitch_contour_reservoir`

`part.create_pitch_contour_reservoir` ()
Creates pitch contour reservoir.

30.2.12 `part.durate_pitch_contour_reservoir`

`part.durate_pitch_contour_reservoir` (*pitch_contour_reservoir*)
Durates pitch contour reservoir.

30.2.13 `part.edit_bass_voice`

`part.edit_bass_voice` (*score*, *durated_reservoir*)
Edits bass voice.

30.2.14 `part.edit_cello_voice`

`part.edit_cello_voice` (*score*, *durated_reservoir*)
Edits cello voice.

30.2.15 `part.edit_first_violin_voice`

`part.edit_first_violin_voice` (*score*, *durated_reservoir*)
Edits first violin voice.

30.2.16 `part.edit_second_violin_voice`

`part.edit_second_violin_voice` (*score*, *durated_reservoir*)
Edits second violin voice.

30.2.17 `part.edit_viola_voice`

`part.edit_viola_voice` (*score*, *durated_reservoir*)
Edits viola voice.

30.2.18 `part.make_part_lilypond_file`

`part.make_part_lilypond_file` ()
Makes Part LilyPond file.

30.2.19 `part.shadow_pitch_contour_reservoir`

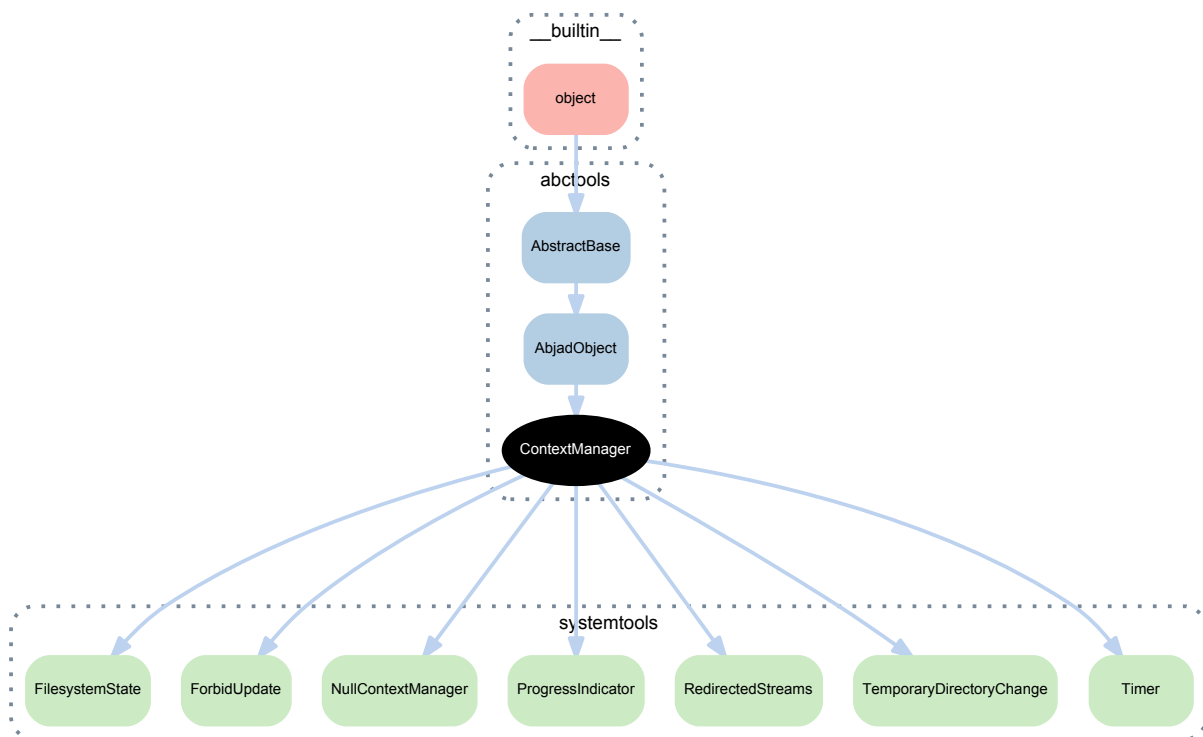
`part.shadow_pitch_contour_reservoir` (*pitch_contour_reservoir*)
Shadows pitch contour reservoir.

Part III

Abjad internal packages

31.1 Abstract classes

31.1.1 abctools.ContextManager



class `abctools.ContextManager`
An abstract context manager class.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

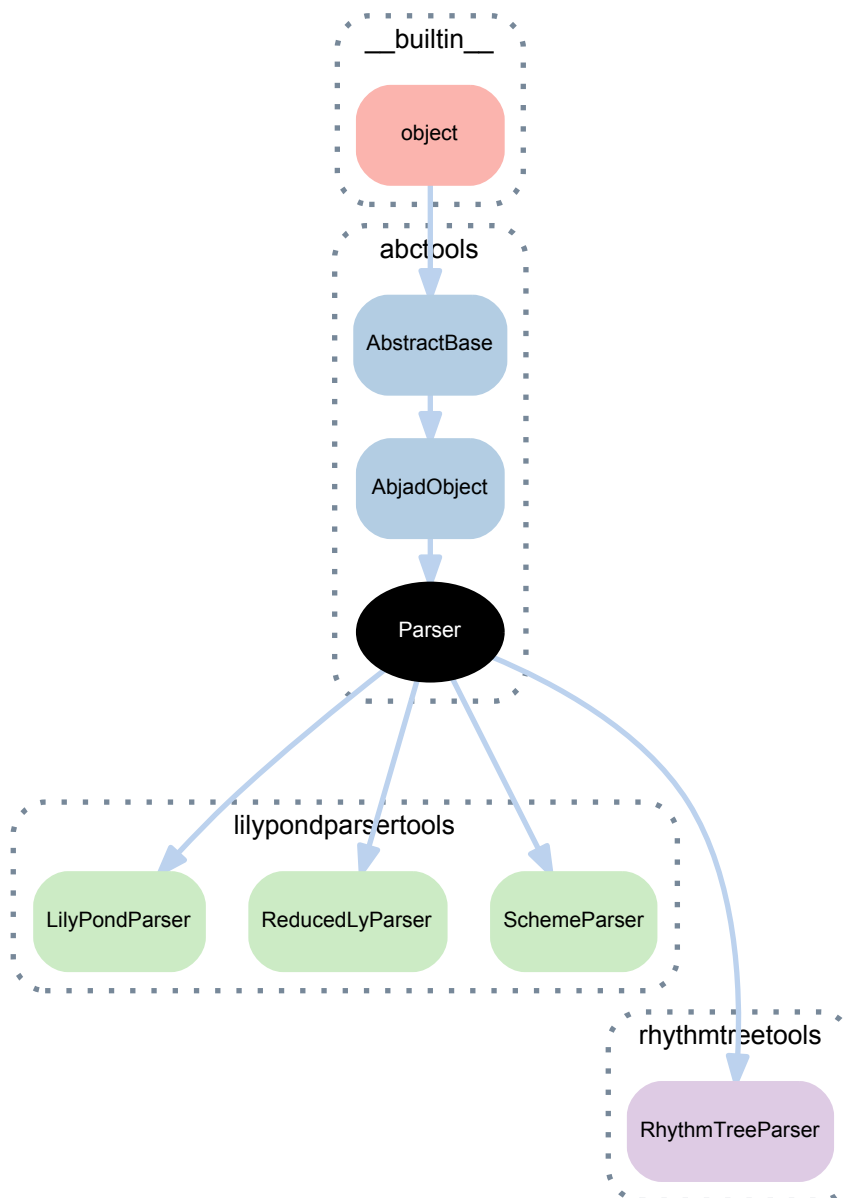
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

31.1.2 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.
- (AbjadObject).__hash__** ()
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

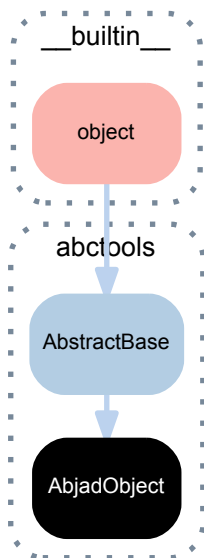
Returns integer.
- (AbjadObject).__ne__** (*expr*)
Is true when Abjad object does not equal *expr*. Otherwise false.

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

Returns string.

31.2 Concrete classes

31.2.1 abctools.AbjadObject



class `abctools.AbjadObject`

Abstract base class from which many custom classes inherit.

Bases

- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`AbjadObject.__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`AbjadObject.__ne__(expr)`

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

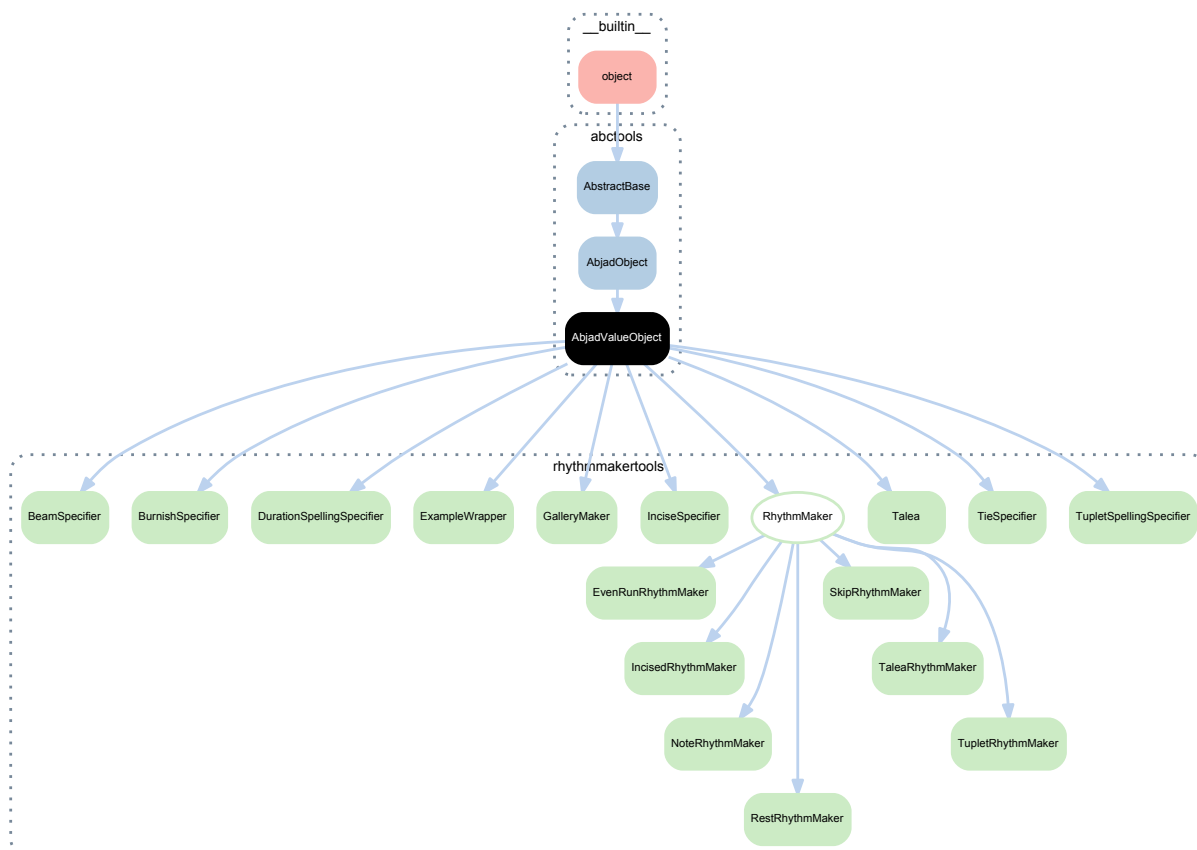
Returns boolean.

`AbjadObject.__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

31.2.2 abctools.AbjadValueObject



class `abctools.AbjadValueObject`

Abstract base class for classes which compare equally based on their storage format.

Bases

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

Special methods

`AbjadValueObject.__eq__(expr)`

Is true when all initialization values of Abjad value object equal the initialization values of *expr*.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

`AbjadValueObject.__hash__()`

Hashes Abjad value object.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

Returns boolean.

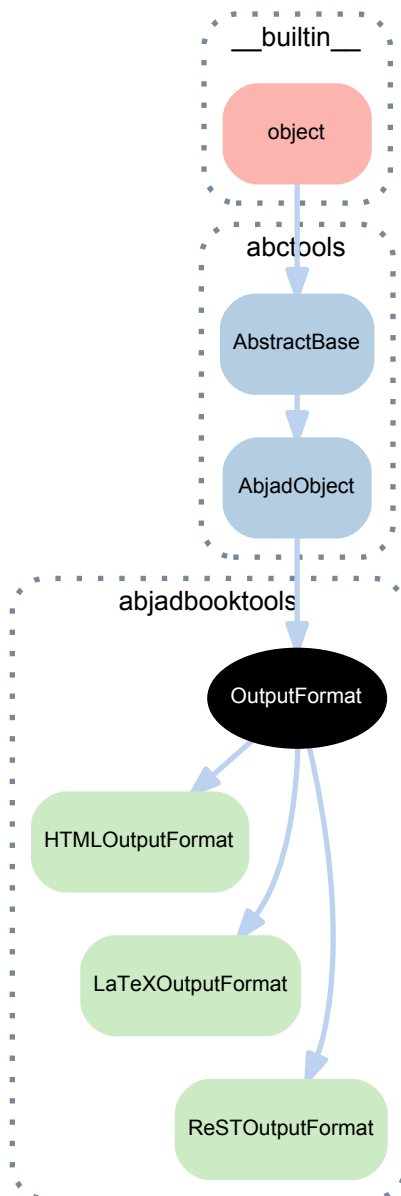
(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

32.1 Abstract classes

32.1.1 abjadbooktools.OutputFormat



```
class abjadbooktools.OutputFormat (code_block_opening, code_block_closing, code_indent,
                                  image_block, image_format)
```

Output format.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

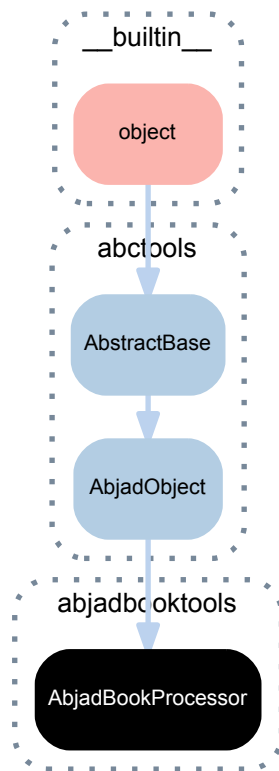
`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
Returns integer.

`(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.

32.2 Concrete classes

32.2.1 abjadbooktools.AbjadBookProcessor



```
class abjadbooktools.AbjadBookProcessor(directory=None, lines=None, out-
                                         put_format=None, skip_rendering=False, im-
                                         age_prefix='image', verbose=False)
```

Abjad book processor.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

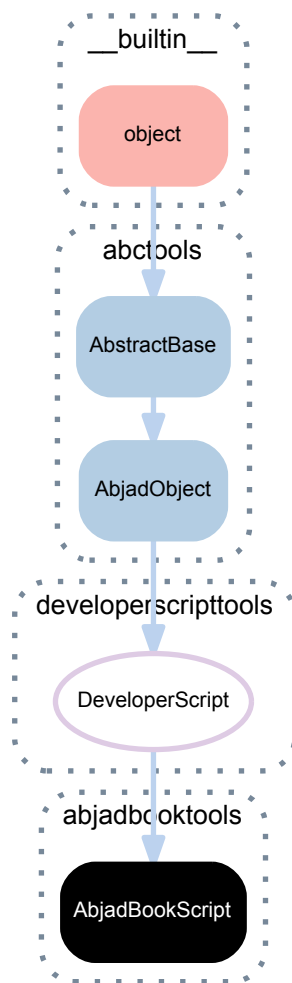
(AbjadObject).**__format__** (*format_specification=''*)
Formats Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
Returns string.

(AbjadObject).**__hash__** ()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

32.2.2 abjadbooktools.AbjadBookScript



class `abjadbooktools.AbjadBookScript`
 abjad book script.

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

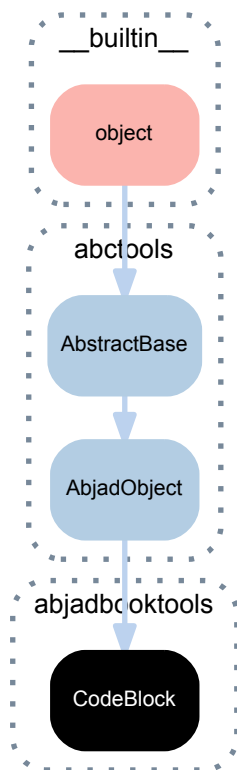
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

32.2.3 abjadbooktools.CodeBlock



class abjadbooktools.**CodeBlock** (*hide=False, lines=None, scale=None, starting_line_number=0, stopping_line_number=1, strip_prompt=False, wrap_width=None*)

A code block.

Bases

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

Read-only properties

`CodeBlock.hide`

Is true when code block should hide.

`CodeBlock.lines`

Lines of code block.

`CodeBlock.processed_results`

Processed results of code block.

`CodeBlock.scale`

Image scaling factor.

`CodeBlock.starting_line_number`

Starting line number of code block.

`CodeBlock.stopping_line_number`

Ending line number of code block.

`CodeBlock.strip_prompt`

Is true when code block should strip prompt.

`CodeBlock.wrap_width`

Wrap width.

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)`

Is 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 Abjad object.

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

Returns string.

`CodeBlock.__hash__()`

Hashes code block.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

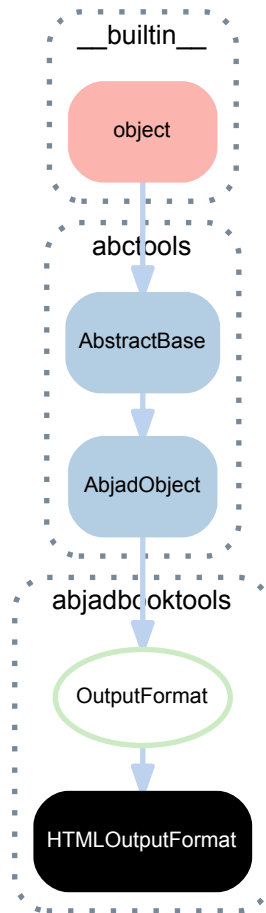
`(AbjadObject).__ne__(expr)`

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

Returns boolean.

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

32.2.4 abjadbooktools.HTMLOutputFormat



class `abjadbooktools.HTMLOutputFormat`
 HTML output format.

Bases

- `abjadbooktools.OutputFormat`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

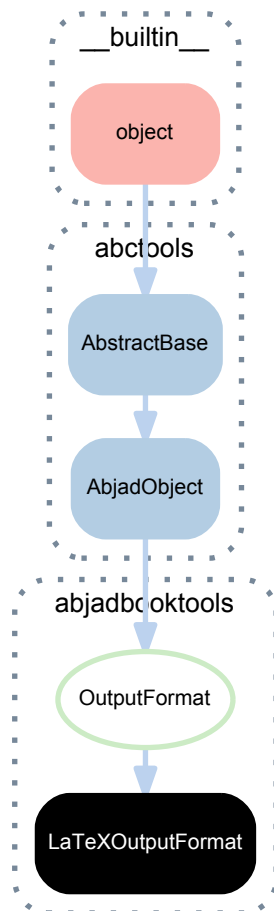
`(AbjadObject).__format__(format_specification='')`
Formats Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

32.2.5 abjadbooktools.LaTeXOutputFormat



class `abjadbooktools.LaTeXOutputFormat`
 LaTeX output format.

Bases

- `abjadbooktools.OutputFormat`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

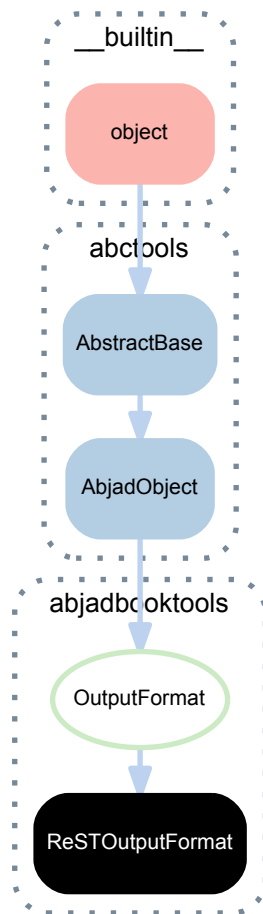
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

32.2.6 abjadbooktools.ReSTOutputFormat



class `abjadbooktools.ReSTOutputFormat`
 ReST output format.

Bases

- `abjadbooktools.OutputFormat`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

Returns boolean.

(AbjadObject) .**__repr__**()

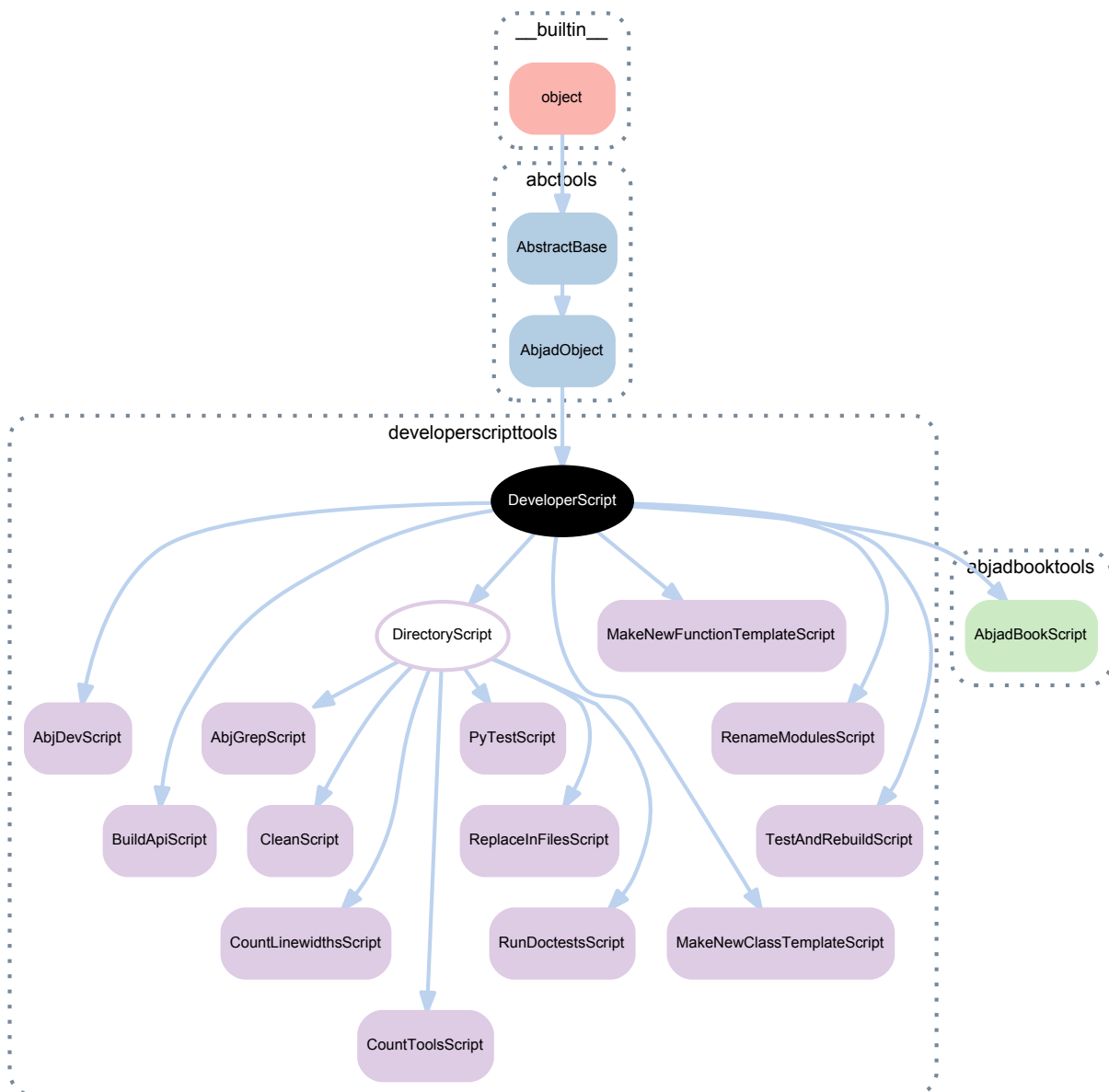
Gets interpreter representation of Abjad object.

Returns string.

DEVELOPERSCRIPTTOOLS

33.1 Abstract classes

33.1.1 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

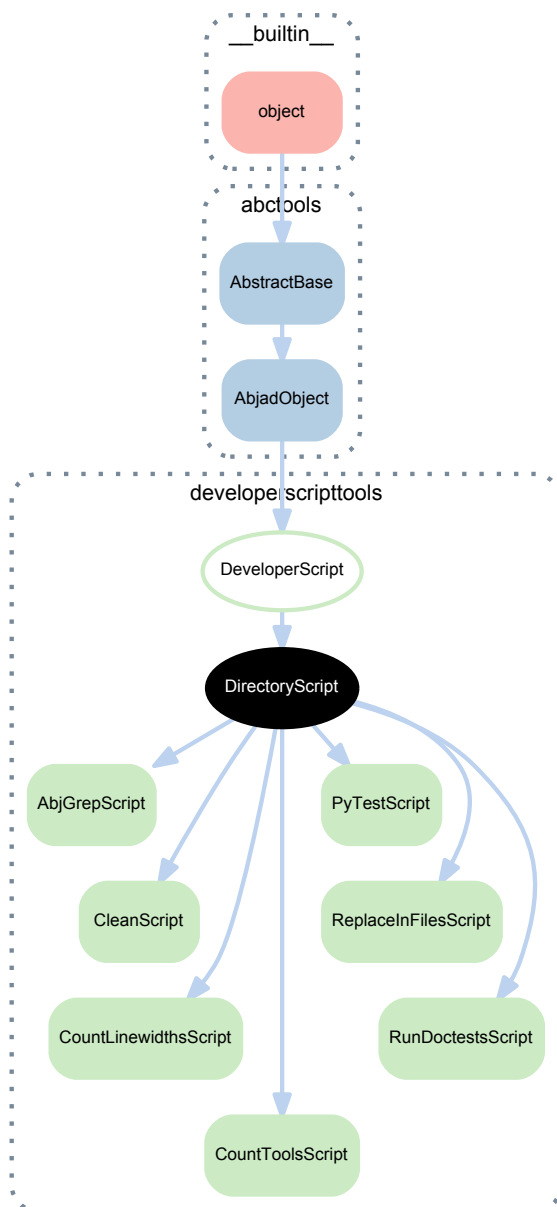
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

33.1.2 developerscripttools.DirectoryScript



class `developerscripttools.DirectoryScript`

DirectoryScript provides utilities for validating file system paths.

DirectoryScript is abstract.

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

Returns boolean.

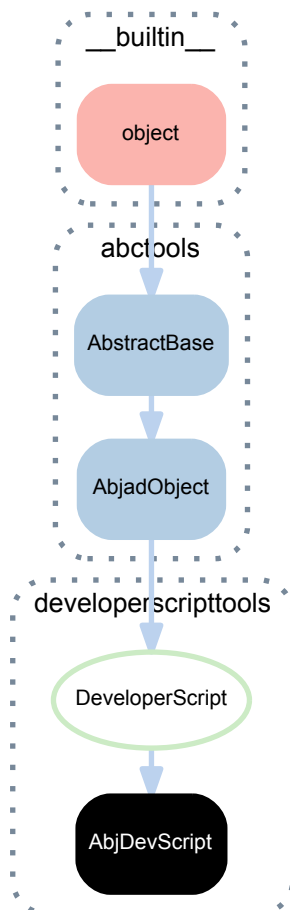
(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.2 Concrete classes

33.2.1 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,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,rename,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
  new                   "new"-related subcommands
  re                    Run pytest -x, doctest -x and then rebuild the API.
  rename                Rename public modules.
  replace               Replace text.
  test                  Run "pytest" on various Abjad paths.

```

ajv supports subcommands similar to *svn*:

```

abjad$ ajv api --help
usage: build-api [-h] [--version] [-M] [-X] [-S] [-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
  -S, --scoremanager     build the Abjad IDE 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

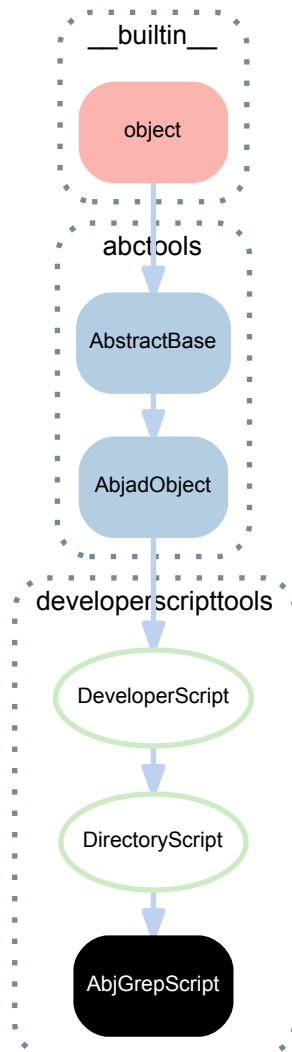
`(AbjadObject).__ne__(expr)`

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

Returns boolean.

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

33.2.2 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

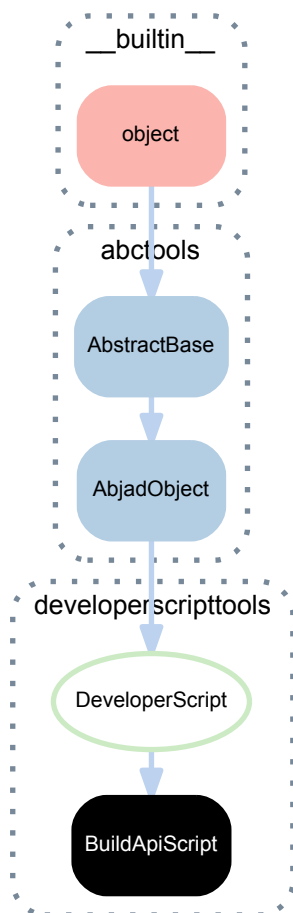
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

33.2.3 developerscripttools.BuildApiScript



class `developerscripttools.BuildApiScript`

Builds the Abjad APIs.

```

abjad$ ajv api --help
usage: build-api [-h] [--version] [-M] [-X] [-S] [-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
  -S, --scoremanager    build the Abjad IDE 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

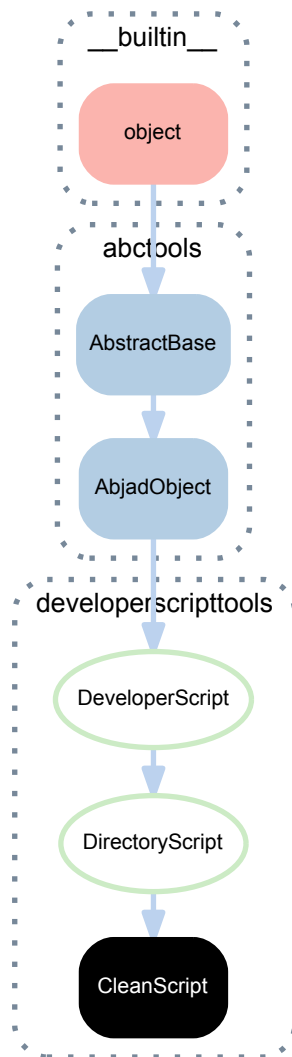
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.2.4 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`

- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

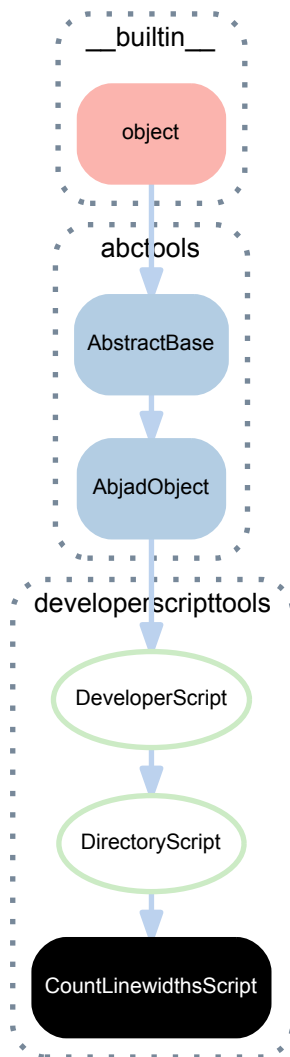
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.2.5 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:

<code>-h, --help</code>	show this help message and exit
<code>--version</code>	show program's version number and exit
<code>-l N, --limit N</code>	limit output to last N items
<code>-o w m, --order-by w m</code>	order by line width [w] or module name [m]
<code>-C, --code</code>	count linewidths of all code in module
<code>-D, --docstrings</code>	count linewidths of all docstrings in module
<code>-a, --ascending</code>	sort results ascending
<code>-d, --descending</code>	sort results descending
<code>-gt N, --greater-than N</code>	line widths greater than N
<code>-lt N, --less-than N</code>	line widths less than N
<code>-eq N, --equal-to N</code>	line widths equal to N

Bases

- `developerscripttools.DirectoryScript`
- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

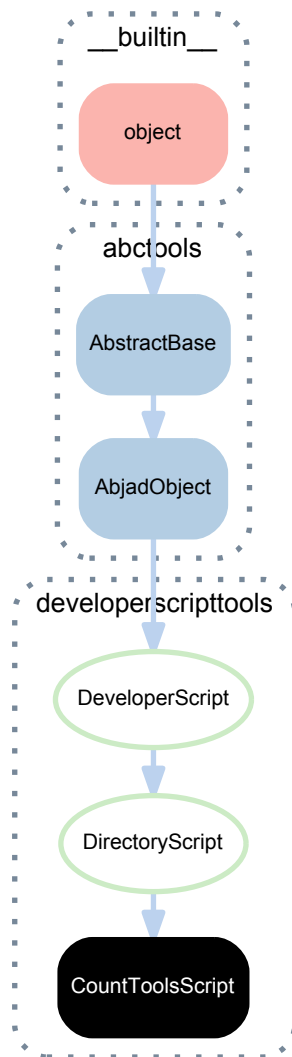
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

33.2.6 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`
- `abctools.AbjadObject.AbstractBase`

- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

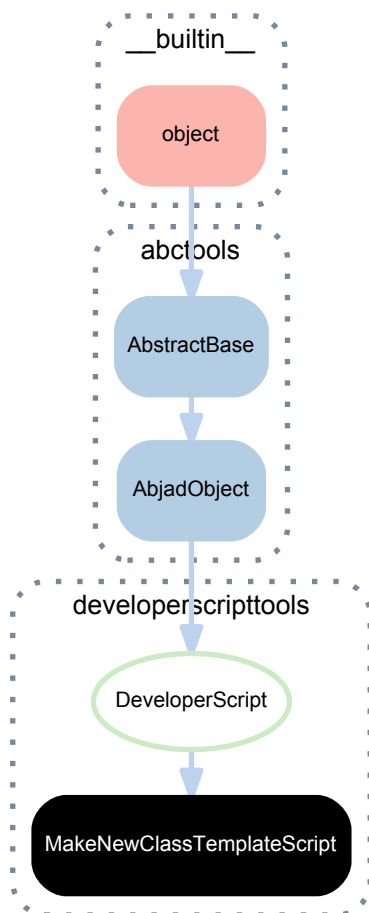
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.2.7 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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

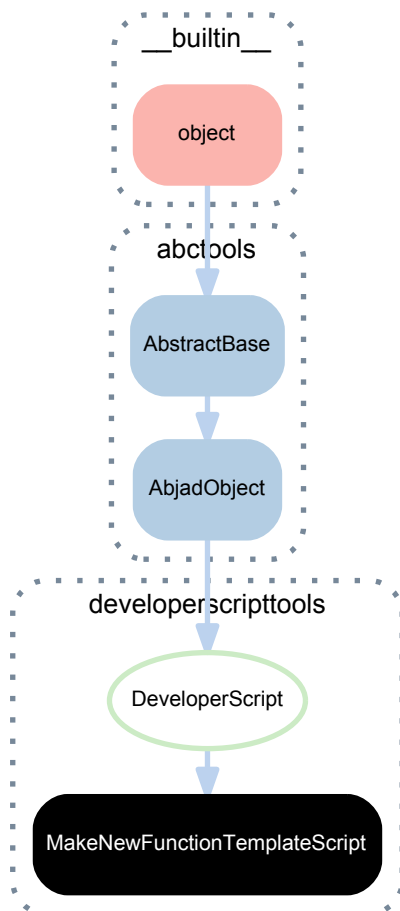
(AbjadObject) .**__format__**(*format_specification*='')
 Formats Abjad object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject) .**__hash__**()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 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.

33.2.8 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

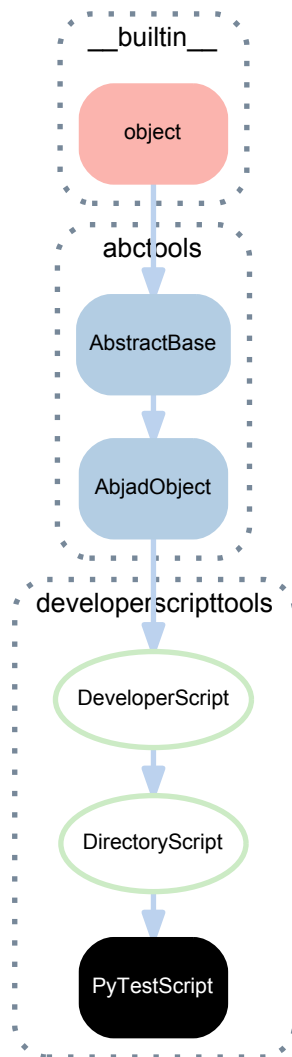
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

33.2.9 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)error, (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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

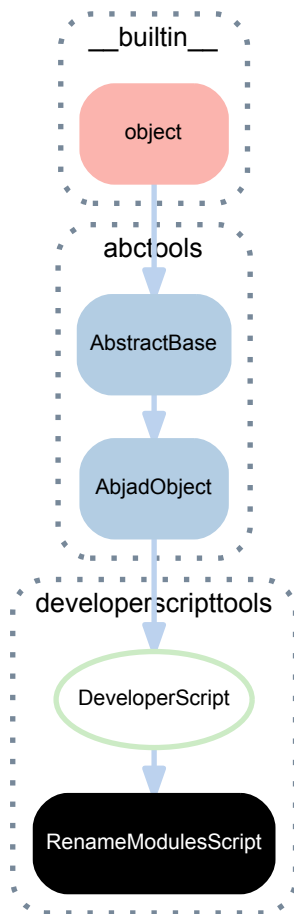
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.2.10 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      toolspackage path of source module
  destination toolspackage 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

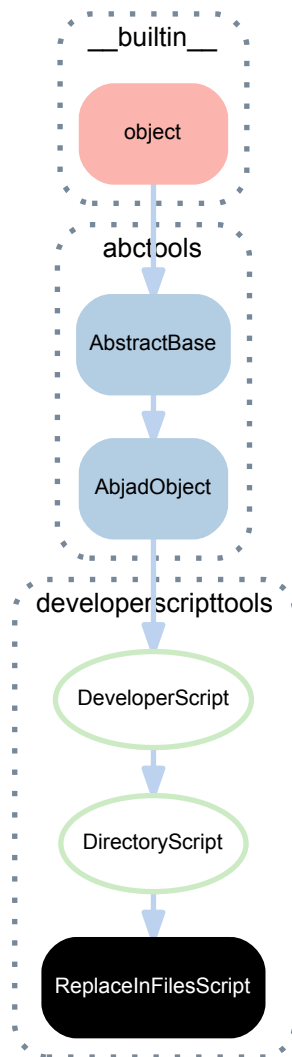
Returns boolean.

(AbjadObject) .**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.2.11 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

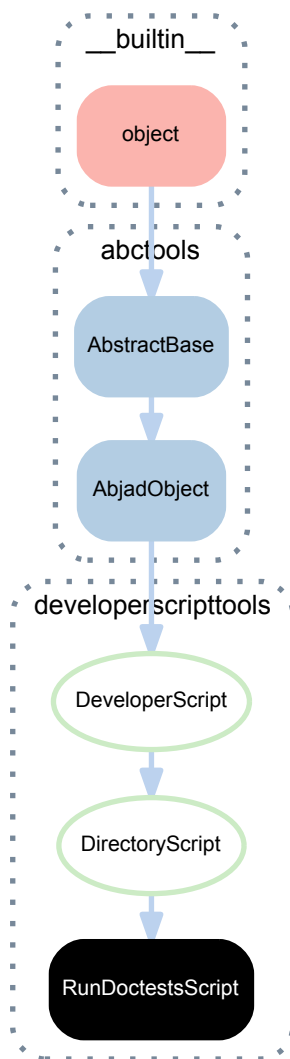
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

33.2.12 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`
- `abctools.AbjadObject.AbstractBase`

- `__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=None, file_paths=None, print_to_terminal=True)`

Processes *args*.

Returns none when *print_to_terminal* is false.

Returns string(s) when *print_to_terminal* is true.

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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

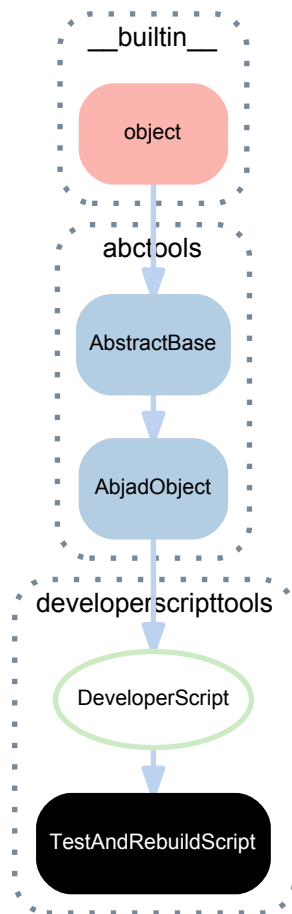
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

33.2.13 developerscripttools.TestAndRebuildScript



class `developerscripttools.TestAndRebuildScript`
 Tests codebase and rebuilds docs.

Bases

- `developerscripttools.DeveloperScript`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

33.3 Functions

33.3.1 `developerscripttools.get_developer_script_classes`

`developerscripttools.get_developer_script_classes()`

Returns a list of all developer script classes.

33.3.2 `developerscripttools.run_abjadbook`

`developerscripttools.run_abjadbook()`

Entry point for `setuptools`.

One-line wrapper around `AbjadBookScript`.

33.3.3 `developerscripttools.run_ajv`

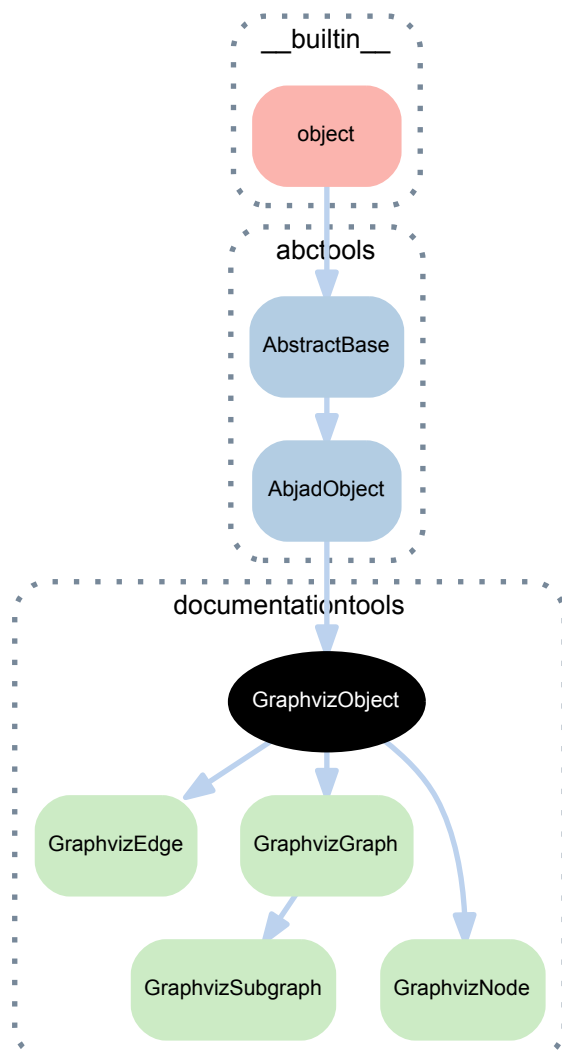
`developerscripttools.run_ajv()`

Entry point for `setuptools`.

One-line wrapper around `AbjDevScript`.

34.1 Abstract classes

34.1.1 documentationtools.GraphvizObject



class `documentationtools.GraphvizObject` (*attributes=None*)
An attributed Graphviz object.

Bases

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

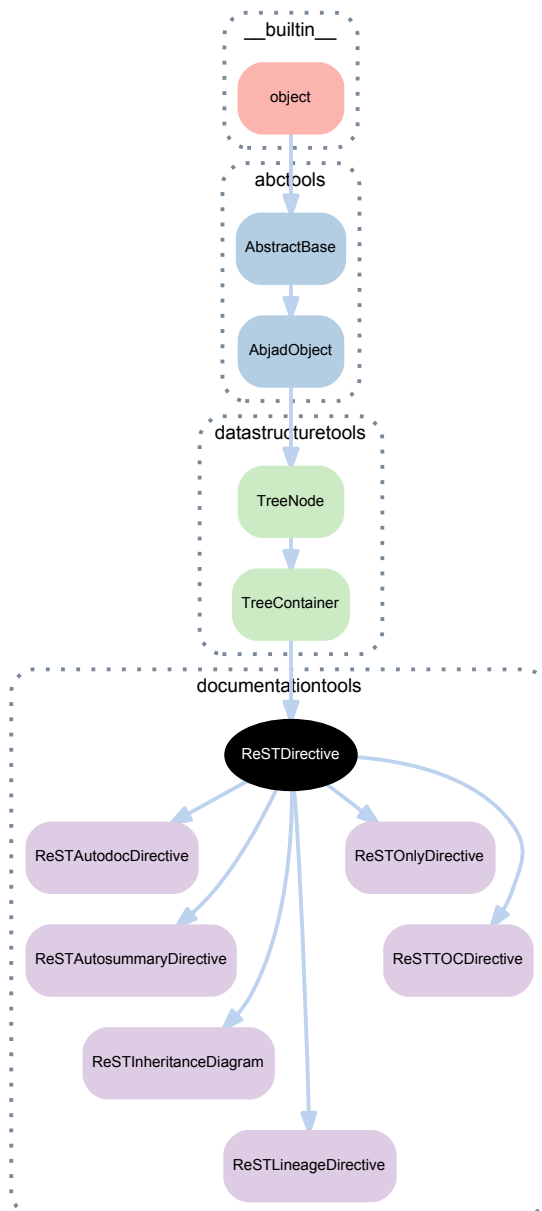
Read-only properties

`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 Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.
- `(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

34.1.2 documentationtools.ReSTDirective



class `documentationtools.ReSTDirective` (*argument=None, children=None, name=None, options=None*)
 A ReST directive.

Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

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 Abjad 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).**__hash__**()
Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()
Iterates tree container.

Yields children of tree container.

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

(TreeNode).**__ne__**(*expr*)
Is 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
```

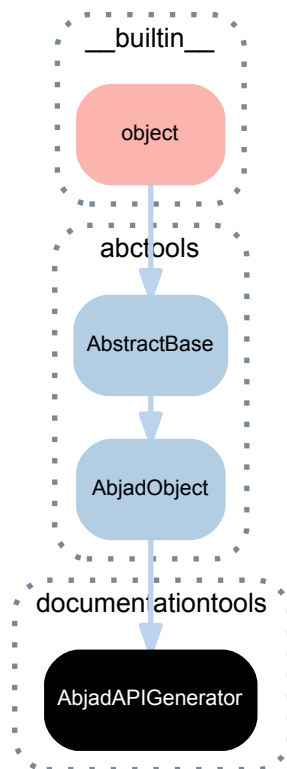
```
>>> b.parent is None
True
```

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

Returns none.

34.2 Concrete classes

34.2.1 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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

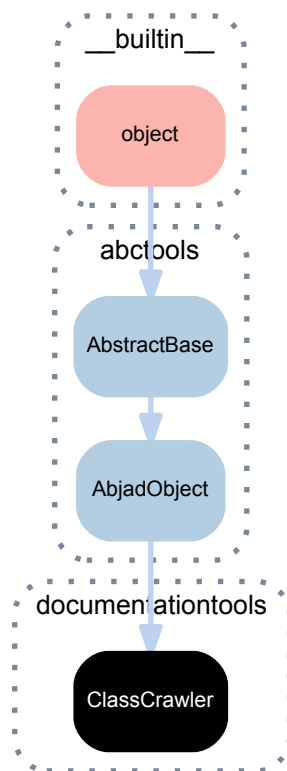
`(AbjadObject).__format__` (*format_specification=''*)
Formats Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__hash__` ()
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

34.2.2 documentationtools.ClassCrawler



class `documentationtools.ClassCrawler` (*code_root=None, include_private_objects=False, root_package_name=None*)
Class crawler.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

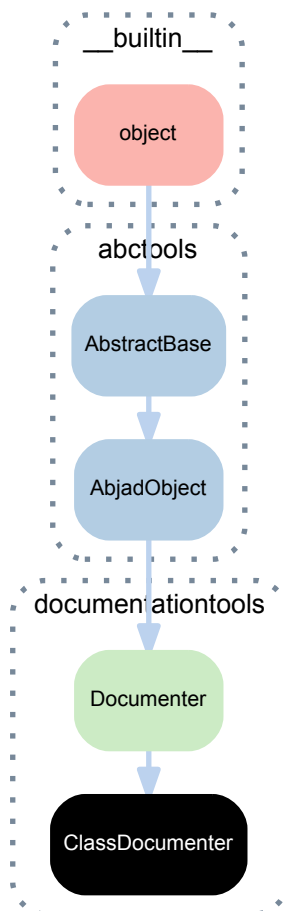
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.2.3 documentationtools.ClassDocumenter



class documentationtools.**ClassDocumenter** (*subject=None, prefix='abjad.tools.'*)

Generates an ReST API entry for a given class.

```
>>> cls = documentationtools.ClassDocumenter
>>> documenter = documentationtools.ClassDocumenter(cls)
>>> restructured_text = documenter()
```

Bases

- `documentationtools.Documenter`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`ClassDocumenter.class_methods`
Class methods.

`ClassDocumenter.data`
Data.

`ClassDocumenter.inherited_attributes`
Inherited attributes.

`ClassDocumenter.is_abstract`
Is true when class is abstract. Otherwise false.

Returns boolean.

`ClassDocumenter.methods`
Methods of class.

`(Documenter).module_name`
Module name of documenter.

Returns string.

`(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.

`(Documenter).subject`
Object of documenter.

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).__format__(format_specification='')`

Formats documenter.

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

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

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

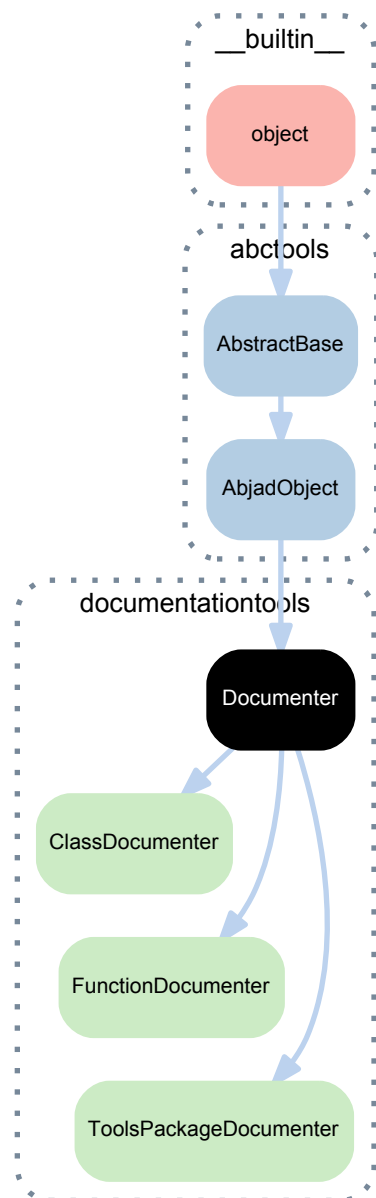
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

34.2.4 documentationtools.Documenter



class `documentationtools.Documenter` (*subject=None, prefix='abjad.tools.'*)
 Documenter is an abstract base class for documentation classes.

Bases

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

Read-only properties

`Documenter.module_name`
 Module name of documenter.
 Returns string.

`Documenter.prefix`

Prefix of documenter.

`Documenter.subject`

Object 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.__format__` (*format_specification*='')

Formats documenter.

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

Returns string.

`(AbjadObject).__hash__` ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

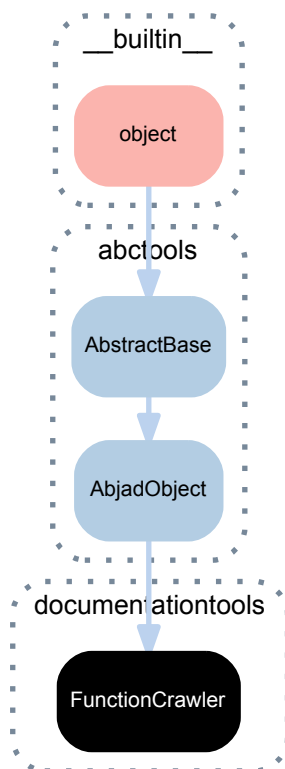
Returns boolean.

`(AbjadObject).__repr__` ()

Gets interpreter representation of Abjad object.

Returns string.

34.2.5 documentationtools.FunctionCrawler



```
class documentationtools.FunctionCrawler (code_root=None, include_private_objects=False, root_package_name=None) in-  
    Function crawler.
```

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

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

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

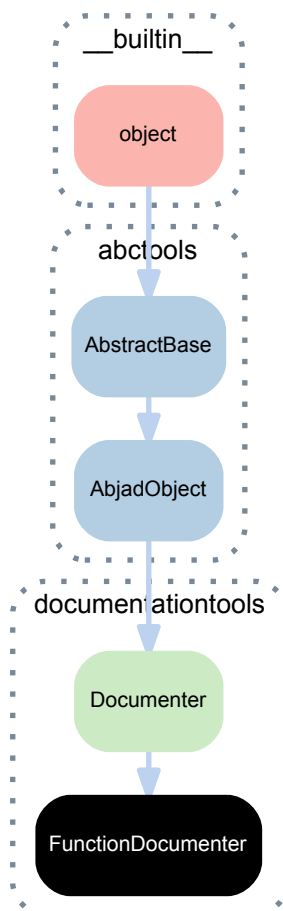
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.2.6 documentationtools.FunctionDocumenter



class `documentationtools.FunctionDocumenter` (*subject=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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`(Documenter).module_name`
 Module name of documenter.
 Returns string.

`(Documenter).prefix`
 Prefix of documenter.

`(Documenter).subject`
 Object 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).__format__(format_specification='')`
 Formats documenter.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

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

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

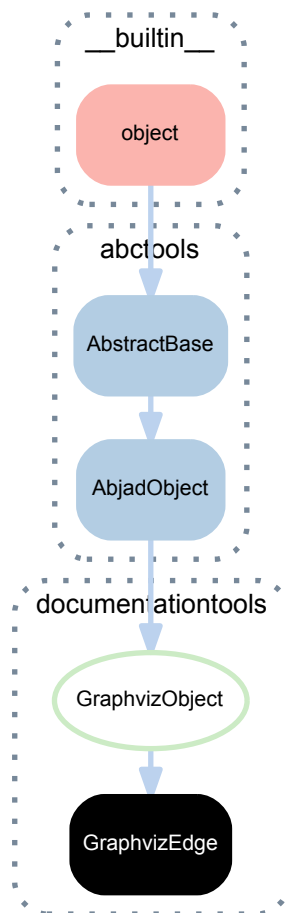
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.2.7 documentationtools.GraphvizEdge



class documentationtools.**GraphvizEdge** (*attributes=None, is_directed=True*)
A Graphviz edge.

Bases

- documentationtools.GraphvizObject
- abctools.AbjadObject
- abctools.AbjadObject.AbstractBase
- __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.**head_port_position**
Head port position.

GraphvizEdge.**is_directed**
Is true when Graphviz edge is directed. Otherwise false.

Returns boolean.

GraphvizEdge.**tail_port_position**
Tail port position.

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 Abjad object.

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

Returns string.

(AbjadObject).**__hash__**()
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

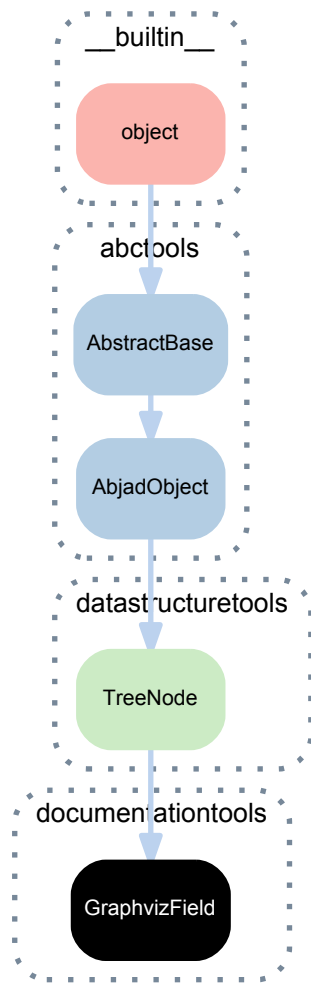
(AbjadObject).**__ne__**(expr)
Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

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

Returns string.

34.2.8 documentationtools.GraphvizField



class `documentationtools.GraphvizField` (*label=None, name=None*)
 A Graphviz struct field.

Bases

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

Read-only properties

`GraphvizField.canonical_name`
 Gets field canonical name.

`(TreeNode).depth`
 The depth of a node in a rhythm-tree structure.

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

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Returns int.

(TreeNode) **.depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node.

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print('DEPTH: {}'.format(depth))
...     for node in inventory[depth]:
...         print(node.name)
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Returns dictionary.

GraphvizField **.edges**

Gets edges of Graphviz struct field.

GraphvizField **.field_name**

Gets the field name.

(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.

GraphvizField.**label**

Gets the field label.

(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.

GraphvizField.**struct**

Gets the parent struct.

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)

Is true when *expr* is a tree node. Otherwise false.

Returns boolean.

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

Formats Abjad object.

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

Returns string.

(TreeNode) **.__hash__**()

Hashes tree node.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TreeNode) **.__ne__**(expr)

Is true when tree node does not equal *expr*. Otherwise false.

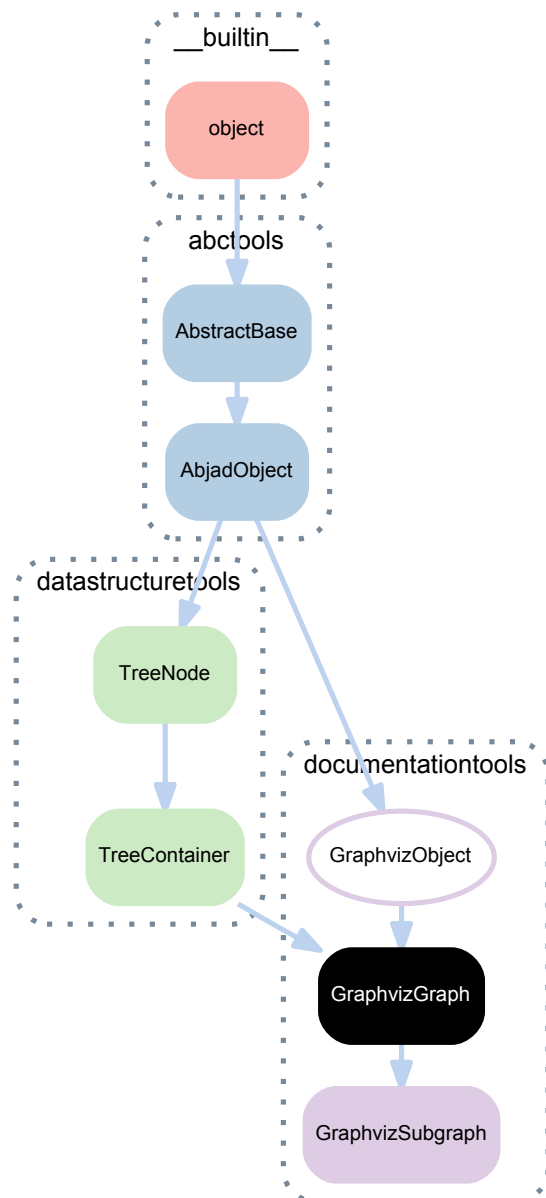
Returns boolean.

(AbjadObject) **.__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.2.9 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;
```

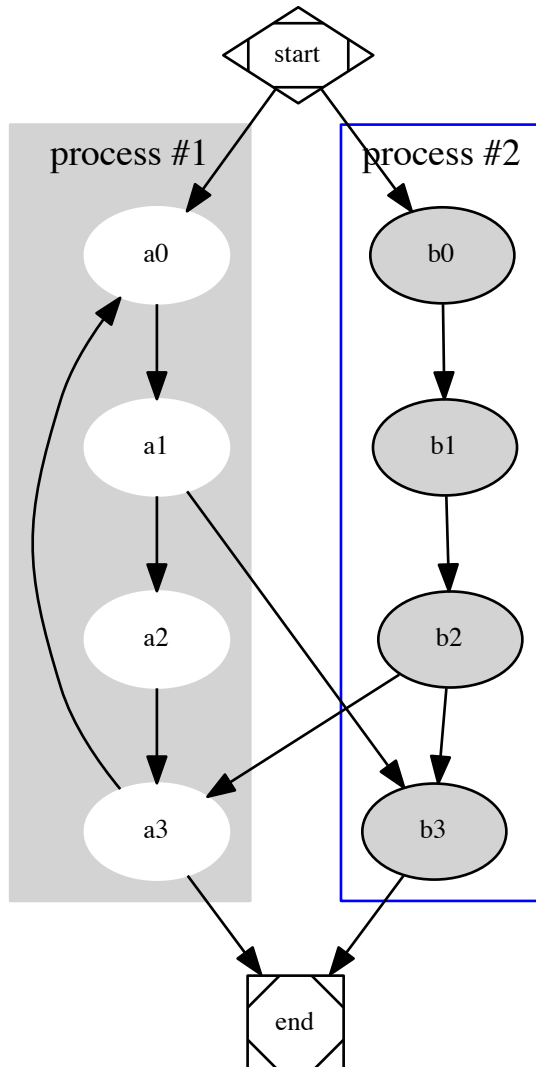
```

    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`
- `abctools.AbjadObject.AbstractBase`
- `__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**

Is 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 Abjad 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).**__hash__**()

Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer) .**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer) .**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode) .**__ne__**(*expr*)

Is 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
```

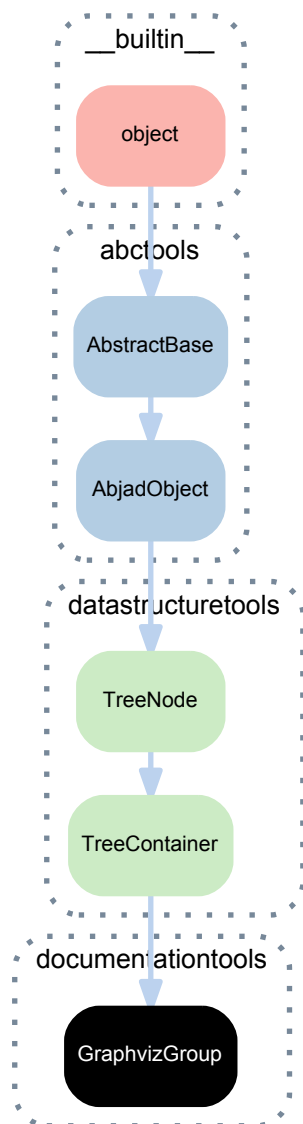
```
>>> c.parent is a
True
```

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

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

Returns none.

34.2.10 documentationtools.GraphvizGroup



class `documentationtools.GraphvizGroup` (*children=None, name=None*)
 A Graphviz struct field grouping.

Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad 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).**__hash__**()
Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()
Iterates tree container.

Yields children of tree container.

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

(TreeNode).**__ne__**(*expr*)
Is 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
```

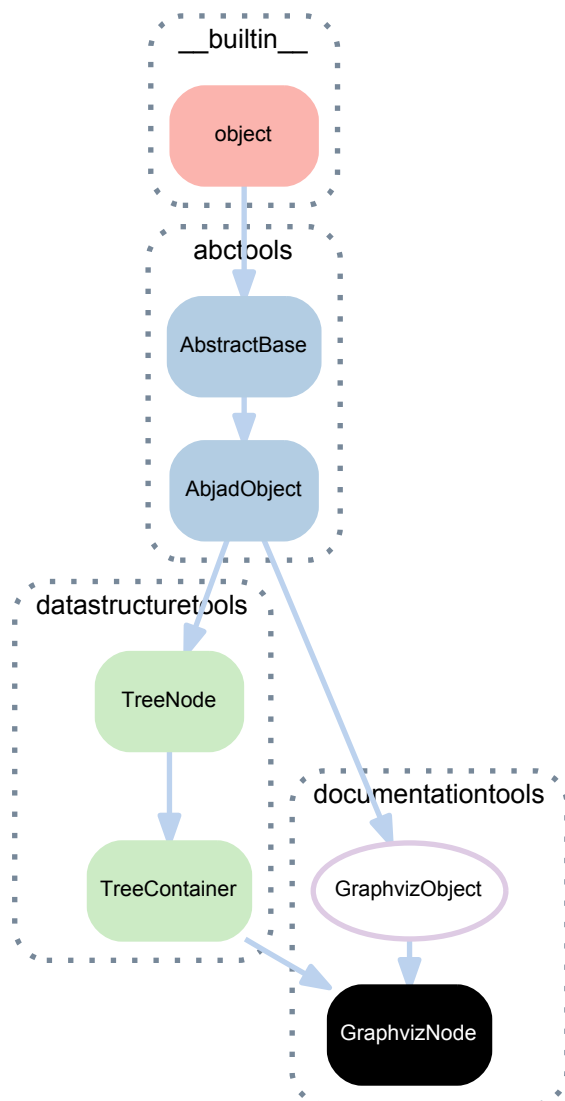
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

34.2.11 documentationtools.GraphvizNode



class `documentationtools.GraphvizNode` (*attributes=None, children=None, name=None*)
 A Graphviz node.

Bases

- `datastructuretools.TreeContainer`
- `datastructuretools.TreeNode`
- `documentationtools.GraphvizObject`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`GraphvizNode.all_edges`

Gets edges of this node and those of any field in its field subtree.

(GraphvizObject).**attributes**
Attributes of Graphviz object.

GraphvizNode.**canonical_name**
Canonical name of Graphviz node.

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.

`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.

`(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 Abjad 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).**__hash__**()
Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()
Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()
Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)
Is 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
```

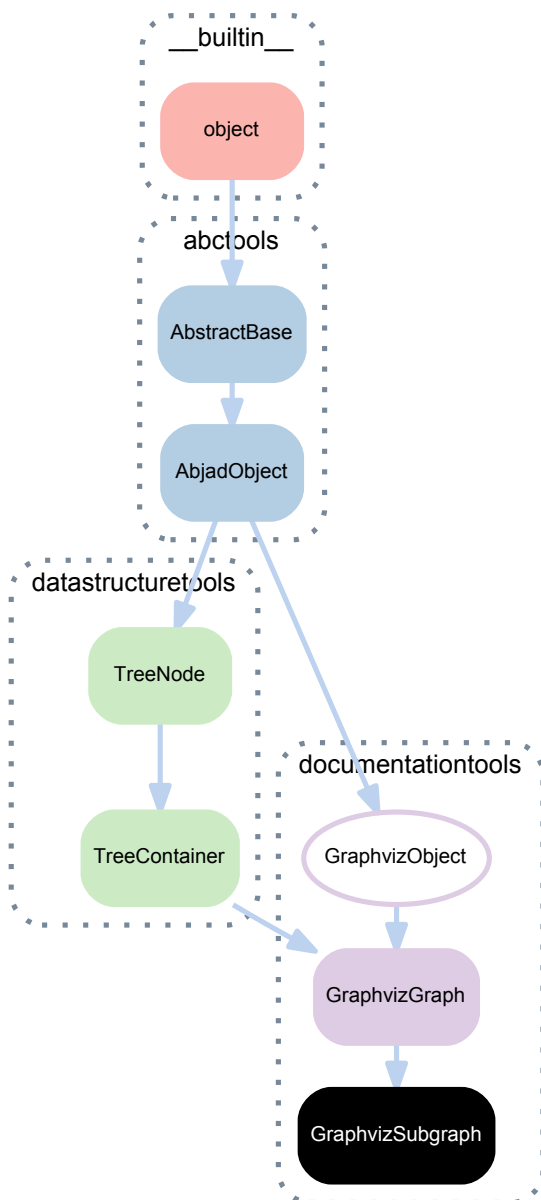
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

34.2.12 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`
- `abctools.AbjadObject.AbstractBase`
- `__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**

Is true when Graphviz subgraph is a cluster. Otherwise false.

Returns boolean.

(GraphvizGraph) **.is_digraph**

Is 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 Abjad 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).**__hash__**()

Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

Is 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
```

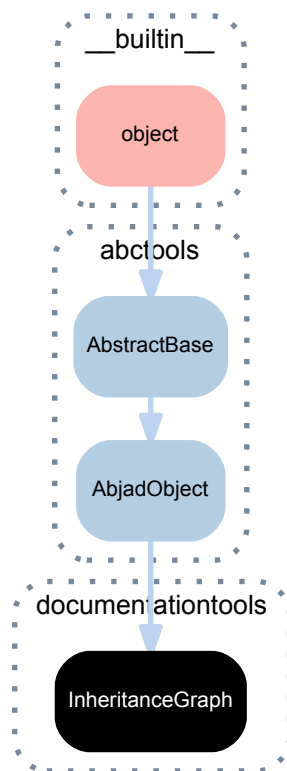
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

34.2.13 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

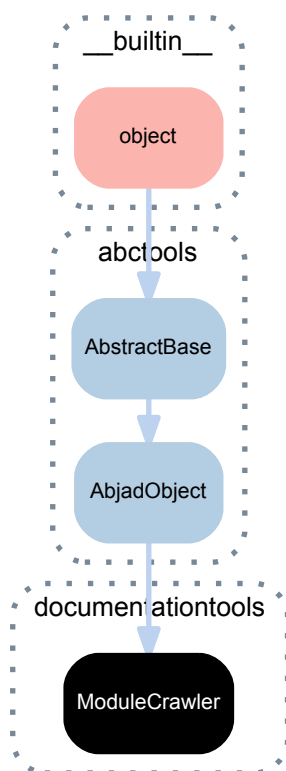
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

34.2.14 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`ModuleCrawler.__iter__(include_score_manager=False)`

Iterates module crawler.

Returns generator.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

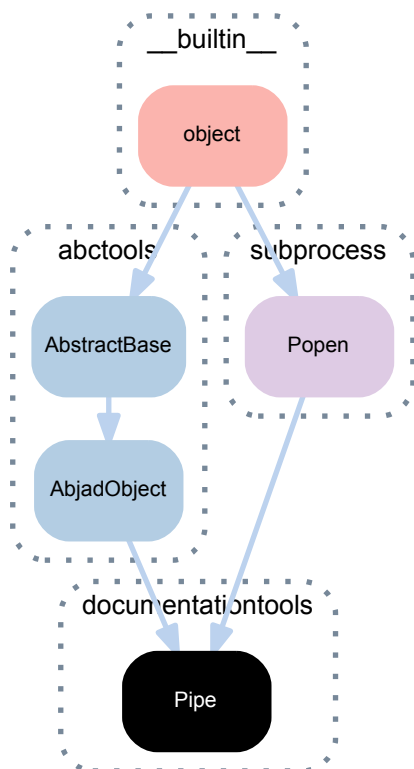
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

34.2.15 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')
```

Bases

- abctools.AbjadObject
- abctools.AbjadObject.AbstractBase
- 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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) **.__hash__** ()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(AbjadObject) **.__ne__** (*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

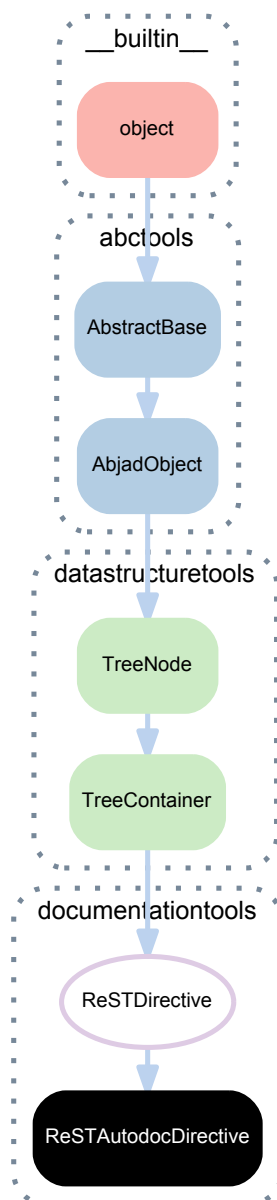
Returns boolean.

(AbjadObject) **.__repr__** ()

Gets interpreter representation of Abjad object.

Returns string.

34.2.16 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`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

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 Abjad 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).**__hash__**()

Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

Is 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
```

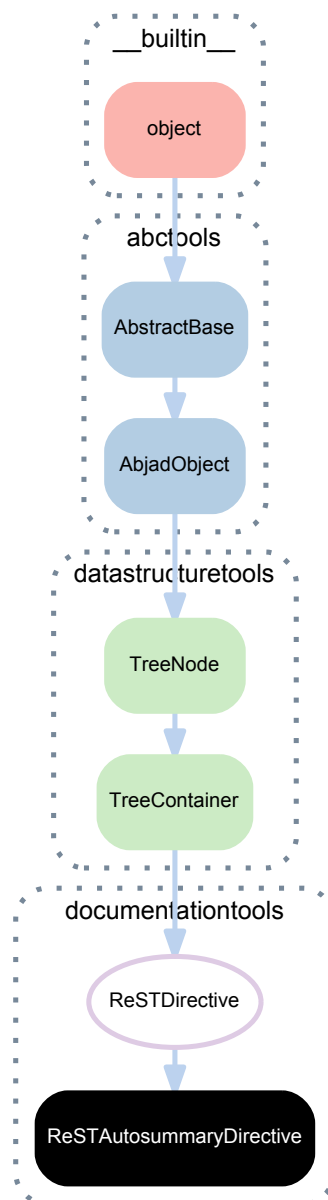
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

34.2.17 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`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

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 Abjad 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).**__hash__**()

Hashes tree container.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

Is true when tree node does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

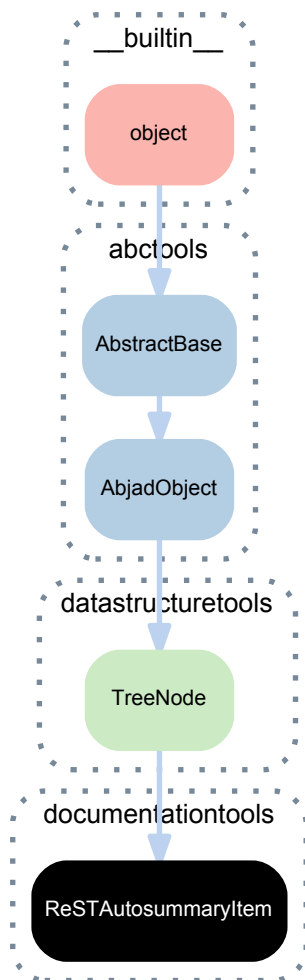
Returns string.

ReSTAutosummaryDirective.**__setitem__**(*i*, *expr*)

Sets item *i* to *expr*.

Returns none.

34.2.18 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`
 Is true when *expr* is a tree node. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`
 Formats Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TreeNode).__hash__()`
 Hashes tree node.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeNode) .__ne__(*expr*)

Is true when tree node does not equal *expr*. Otherwise false.

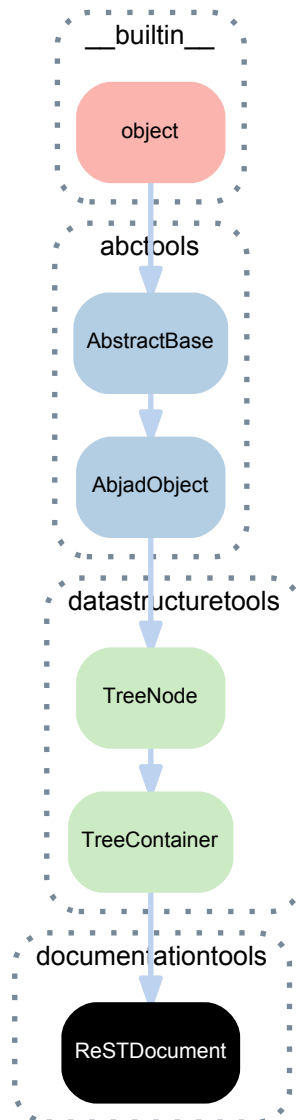
Returns boolean.

(AbjadObject) .__repr__()

Gets interpreter representation of Abjad object.

Returns string.

34.2.19 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad 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).**__hash__**()
Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()
Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()
Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)
Is 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
```

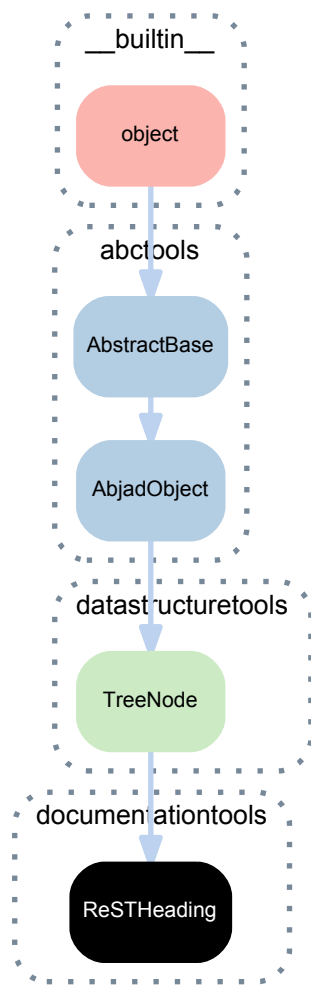
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

34.2.20 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is true when *expr* is a tree node. Otherwise false.

Returns boolean.

`(AbjadObject).__format__(format_specification='')`

Formats Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(TreeNode).__hash__()`

Hashes tree node.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

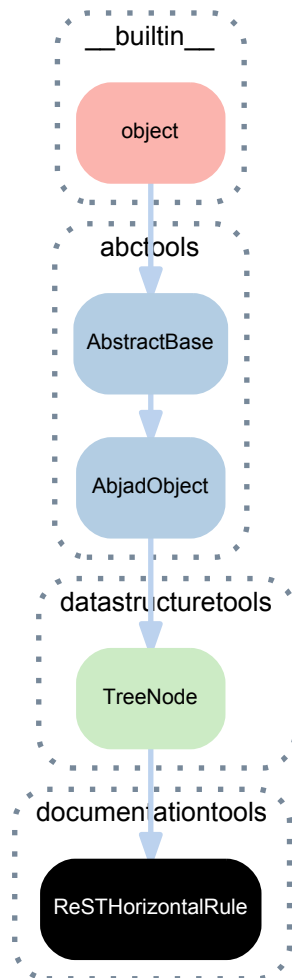
`(TreeNode).__ne__(expr)`

Is true when tree node does not equal *expr*. Otherwise false.

Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

34.2.21 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
- abctools.AbjadObject.AbstractBase
- __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)
 Is true when *expr* is a tree node. Otherwise false.
 Returns boolean.

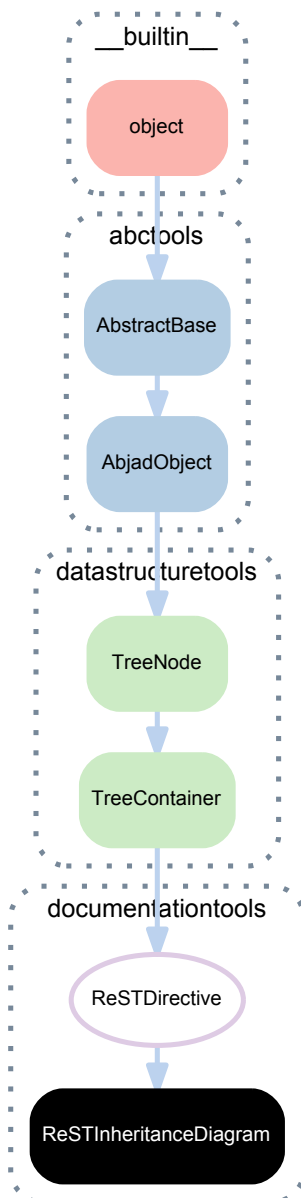
(AbjadObject) **.__format__** (format_specification='')
 Formats Abjad object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(TreeNode) **.__hash__** ()
 Hashes tree node.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 Returns integer.

(TreeNode) **.__ne__** (expr)
 Is true when tree node does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject) **.__repr__** ()
 Gets interpreter representation of Abjad object.
 Returns string.

34.2.22 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad 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).**__hash__**()
Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer).**__iter__**()
Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()
Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)
Is 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
```

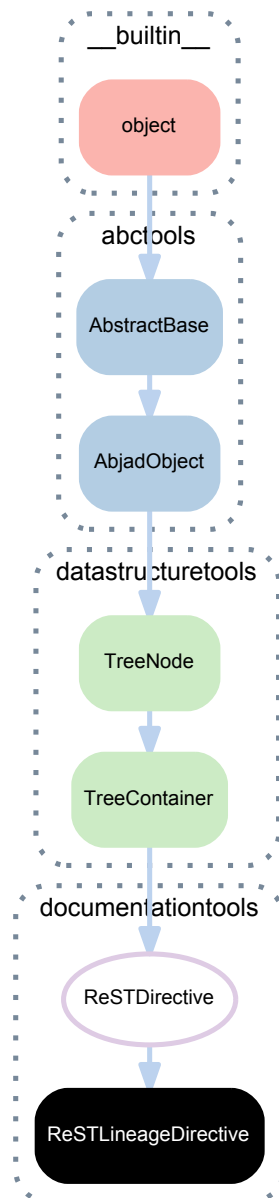
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

34.2.23 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`
- `abctools.AbjadObject.AbstractBase`
- `__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*)

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 Abjad 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).**__hash__**()

Hashes tree container.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(TreeContainer).**__iter__**()

Iterates tree container.

Yields children of tree container.

(TreeContainer).**__len__**()

Returns nonnegative integer number of nodes in container.

(TreeNode).**__ne__**(*expr*)

Is 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
```

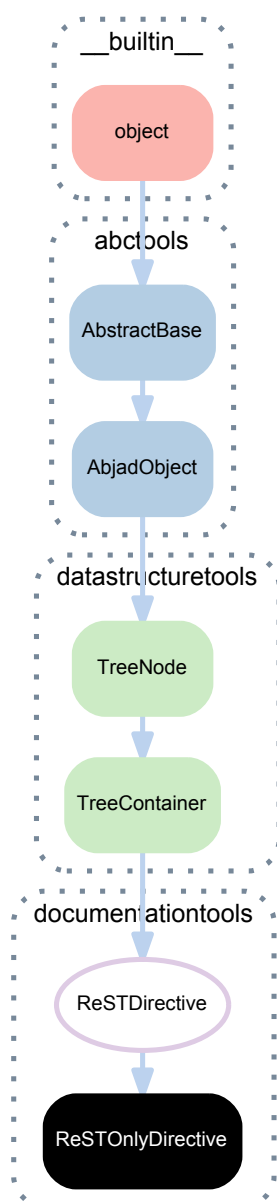
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

34.2.24 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad 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) .**__hash__** ()
 Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(TreeContainer) .**__iter__** ()
 Iterates tree container.

Yields children of tree container.

(TreeContainer) .**__len__** ()
 Returns nonnegative integer number of nodes in container.

(TreeNode) .**__ne__** (*expr*)
 Is 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
```

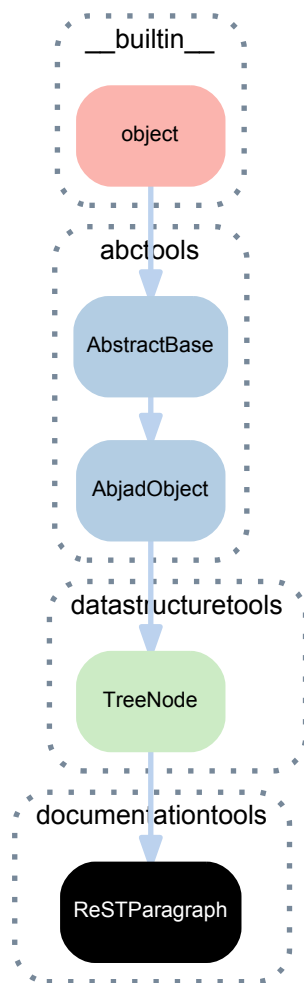
```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Returns none.

34.2.25 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`
- `abctools.AbjadObject.AbstractBase`
- `___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)

Is true when *expr* is a tree node. Otherwise false.

Returns boolean.

(AbjadObject) **.__format__** (format_specification='')

Formats Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(TreeNode) **.__hash__** ()

Hashes tree node.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

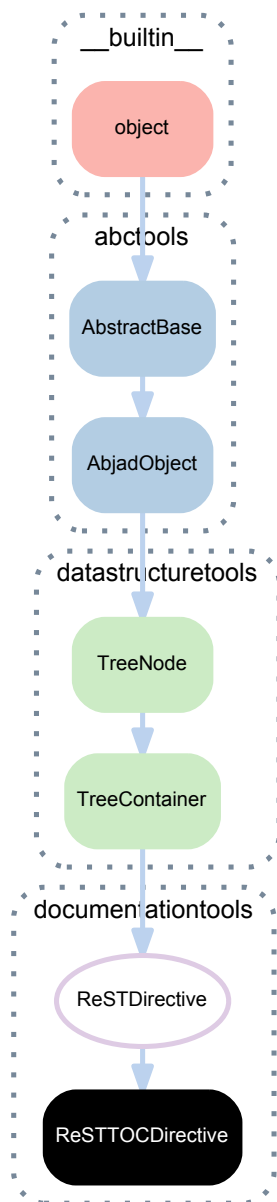
(TreeNode) **.__ne__** (expr)

Is true when tree node does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`
 Gets interpreter representation of Abjad object.
 Returns string.

34.2.26 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad 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).__hash__()`

Hashes tree container.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(TreeContainer).__iter__()`

Iterates tree container.

Yields children of tree container.

`(TreeContainer).__len__()`

Returns nonnegative integer number of nodes in container.

`(TreeNode).__ne__(expr)`

Is true when tree node does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

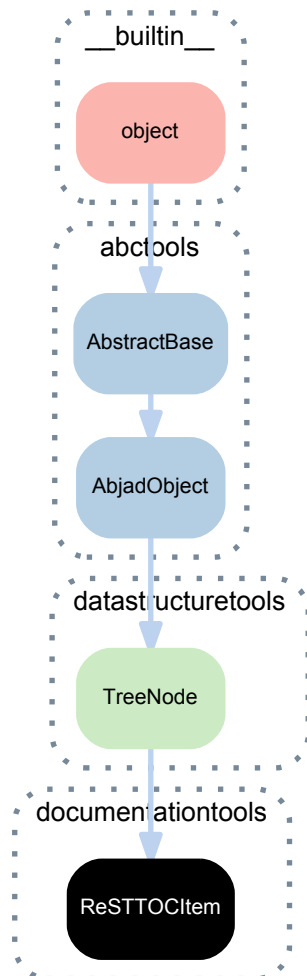
Returns string.

`ReSTTOCDirective.__setitem__(i, expr)`

Sets *i* to *expr*.

Returns none.

34.2.27 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`
- `abctools.AbjadObject.AbstractBase`
- `__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)`

Is true when *expr* is a tree node. Otherwise false.

Returns boolean.

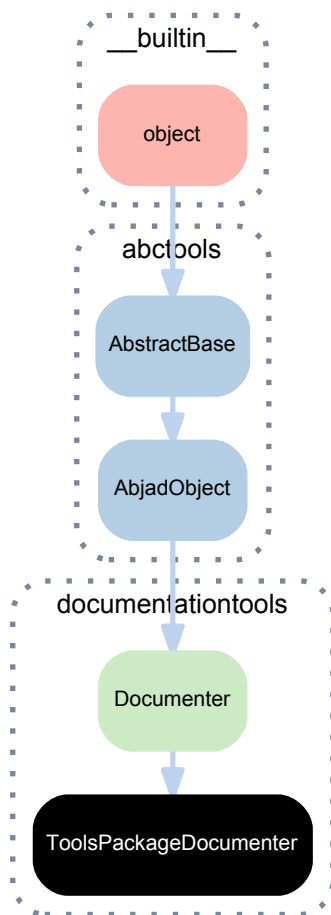
(AbjadObject). **__format__** (*format_specification*='')
 Formats Abjad object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(TreeNode). **__hash__** ()
 Hashes tree node.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 Returns integer.

(TreeNode). **__ne__** (*expr*)
 Is true when tree node does not equal *expr*. Otherwise false.
 Returns boolean.

(AbjadObject). **__repr__** ()
 Gets interpreter representation of Abjad object.
 Returns string.

34.2.28 documentationtools.ToolsPackageDocumenter



class `documentationtools.ToolsPackageDocumenter` (*subject=None*, *nored_directory_names=()*, *ig-pre-fix='abjad.tools.'*)

Generates an index for every tools package.

Bases

- `documentationtools.Documenter`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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).prefix`
Prefix of documenter.

`(Documenter).subject`
Object 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).__format__(format_specification='')`

Formats documenter.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

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.

34.3 Functions

34.3.1 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.

34.3.2 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()
```

34.3.3 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()
```

34.3.4 documentationtools.list_all_classes

`documentationtools.list_all_classes (modules=None)`

Lists all public classes defined in *path*.

```
>>> all_classes = documentationtools.list_all_classes(  
...     modules='experimental',  
... )
```

34.3.5 documentationtools.list_all_experimental_classes

`documentationtools.list_all_experimental_classes (modules=None)`

Lists all public classes defined in the Abjad experimental branch.

```
>>> all_classes = documentationtools.list_all_experimental_classes()
```

34.3.6 documentationtools.list_all_scoremanager_classes

`documentationtools.list_all_scoremanager_classes (modules=None)`

Lists all public classes defined in Abjad.

```
>>> all_classes = documentationtools.list_all_scoremanager_classes()
```

34.3.7 documentationtools.list_all_scoremanager_functions

`documentationtools.list_all_scoremanager_functions (modules=None)`

Lists all public functions defined in Abjad.

```
>>> all_functions = documentationtools.list_all_scoremanager_functions()
```

34.3.8 documentationtools.make_ligeti_example_lilypond_file

`documentationtools.make_ligeti_example_lilypond_file (music=None)`

Makes Ligeti example LilyPond file.

Returns LilyPond file.

34.3.9 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.

34.3.10 `documentationtools.make_reference_manual_lilypond_file`

`documentationtools.make_reference_manual_lilypond_file` (*music=None*)

Makes reference manual LilyPond file.

```
>>> score = Score([Staff('c d e f')])
>>> lilypond_file = \
...     documentationtools.make_reference_manual_lilypond_file(score)
```

Returns LilyPond file.

34.3.11 `documentationtools.make_text_alignment_example_lilypond_file`

`documentationtools.make_text_alignment_example_lilypond_file` (*music=None*)

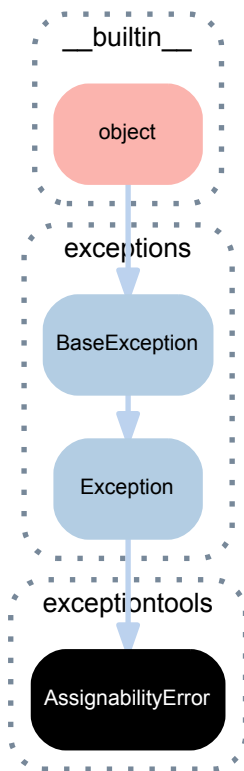
Makes text-alignment example LilyPond file.

```
>>> score = Score([Staff('c d e f')])
>>> lilypond_file = documentationtools.make_text_alignment_example_lilypond_file(score)
```

Returns LilyPond file.

35.1 Concrete classes

35.1.1 `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.

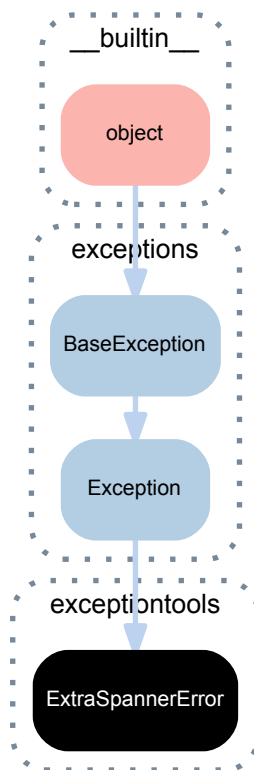
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

35.1.2 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.

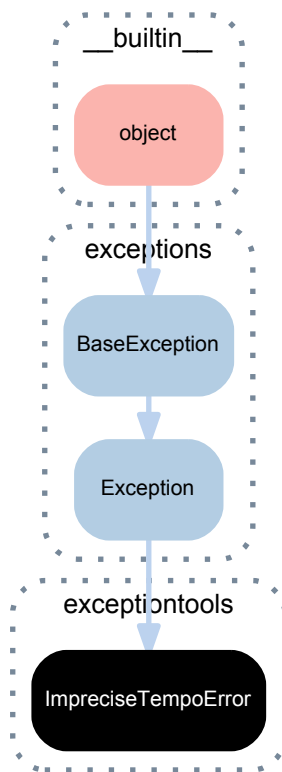
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

35.1.3 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.

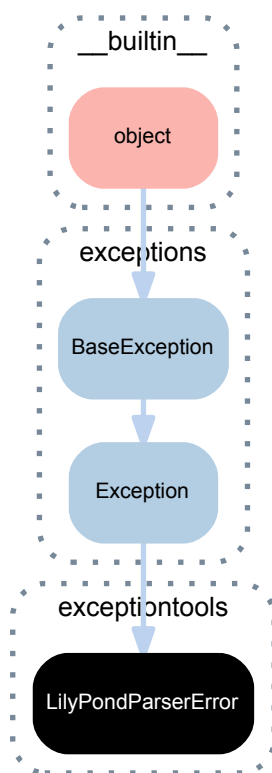
```
(BaseException).__repr__() <==> repr(x)
```

```
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException).__str__() <==> str(x)
```

```
(BaseException).__unicode__()
```

35.1.4 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.

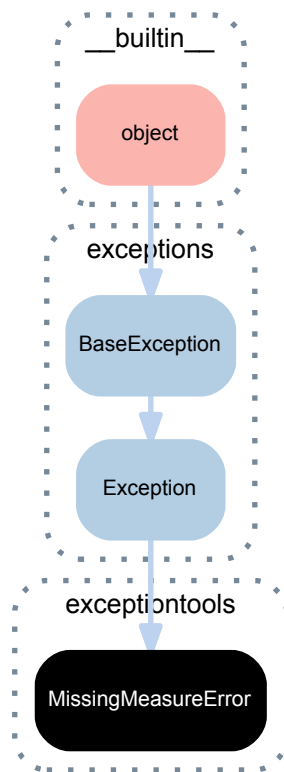
```
(BaseException).__repr__() <==> repr(x)
```

```
(BaseException).__setattr__()
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException).__str__() <==> str(x)
```

```
(BaseException).__unicode__()
```

35.1.5 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.

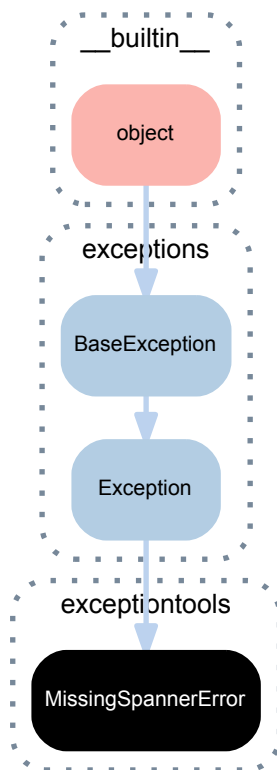
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

35.1.6 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.

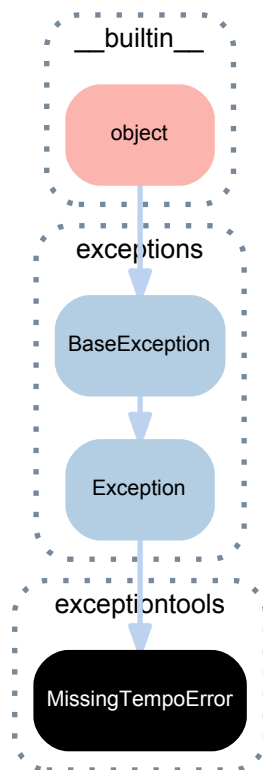
```
(BaseException).__repr__() <==> repr(x)
```

```
(BaseException).__setattr__()
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException).__str__() <==> str(x)
```

```
(BaseException).__unicode__()
```

35.1.7 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.

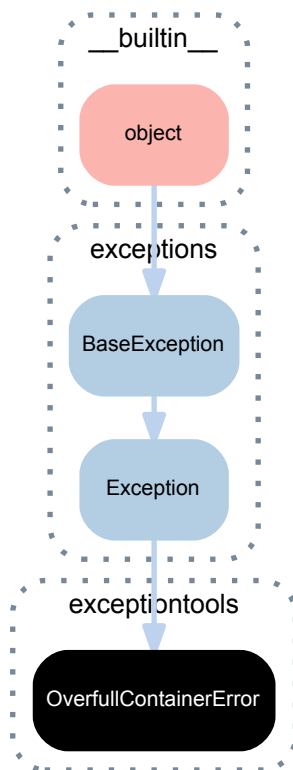
```
(BaseException).__repr__() <==> repr(x)
```

```
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException).__str__() <==> str(x)
```

```
(BaseException).__unicode__()
```

35.1.8 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.

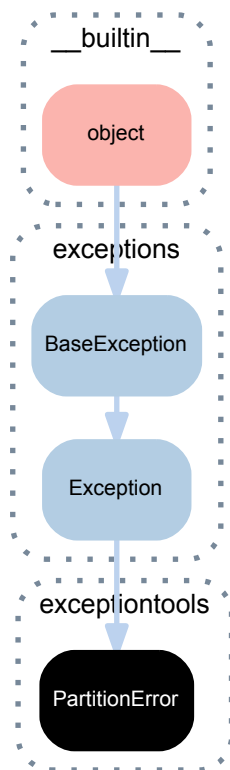
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

35.1.9 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.

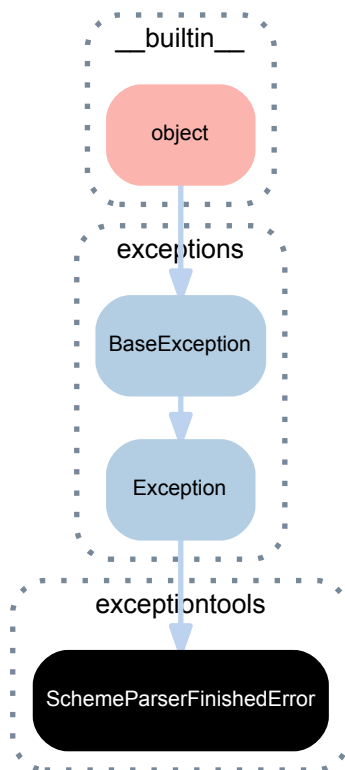
```
(BaseException).__repr__() <==> repr(x)
```

```
(BaseException).__setattr__()  
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException).__str__() <==> str(x)
```

```
(BaseException).__unicode__()
```

35.1.10 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.

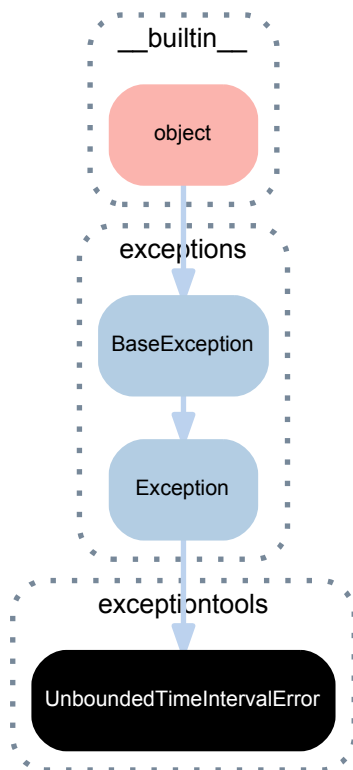
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

35.1.11 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.

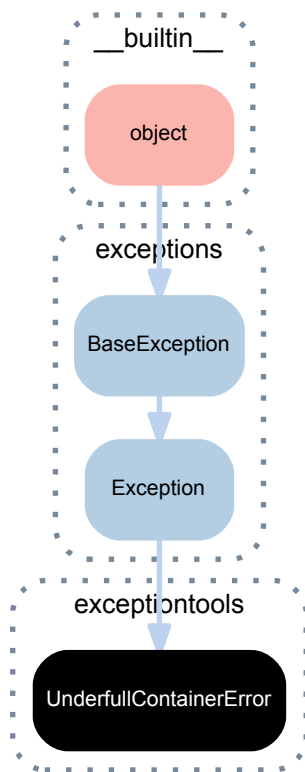
(BaseException).__repr__() <==> repr(x)

(BaseException).__setattr__()
    x.__setattr__('name', value) <==> x.name = value

(BaseException).__str__() <==> str(x)

(BaseException).__unicode__()
```

35.1.12 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)
```

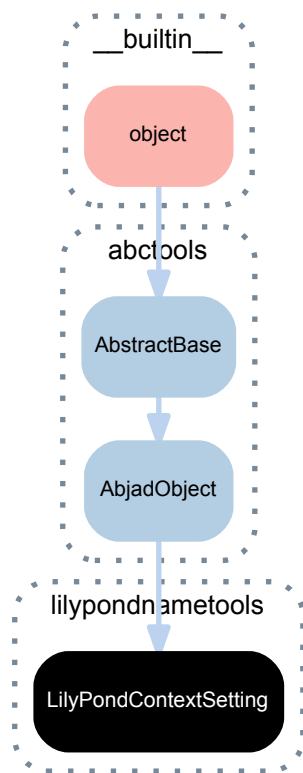
```
(BaseException) .__setattr__()  
x.__setattr__('name', value) <==> x.name = value
```

```
(BaseException) .__str__() <==> str(x)
```

```
(BaseException) .__unicode__()
```


36.1 Concrete classes

36.1.1 lilypondnametools.LilyPondContextSetting



```
class lilypondnametools.LilyPondContextSetting(context_name=None,          con-
                                              text_property='autoBeaming',
                                              is_unset=False, value=False)
```

A LilyPond context setting.

```
>>> context_setting = lilypondnametools.LilyPondContextSetting(
...     context_name='Score',
...     context_property='autoBeaming',
...     value=False,
... )
```

```
>>> print('\n'.join(context_setting.format_pieces))
\set Score.autoBeaming = ##f
```

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`LilyPondContextSetting.context_name`

Optional LilyPond context name.

Returns string or none.

`LilyPondContextSetting.context_property`

LilyPond context property name.

Returns string.

`LilyPondContextSetting.format_pieces`

Gets LilyPond context setting set or unset format pieces.

Returns tuple of strings.

`LilyPondContextSetting.is_unset`

Is true if context setting unsets its value. Otherwise false.

Returns boolean or none.

`LilyPondContextSetting.value`

Value of LilyPond context setting.

Returns arbitrary object.

Special methods

`LilyPondContextSetting.__eq__(expr)`

Is true when *expr* is a LilyPond context setting with equivalent keyword values.

`(AbjadObject).__format__(format_specification='')`

Formats Abjad object.

Set *format_specification* to `''` or `'storage'`. Interprets `''` equal to `'storage'`.

Returns string.

`LilyPondContextSetting.__hash__()`

Hashes LilyPond context setting.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

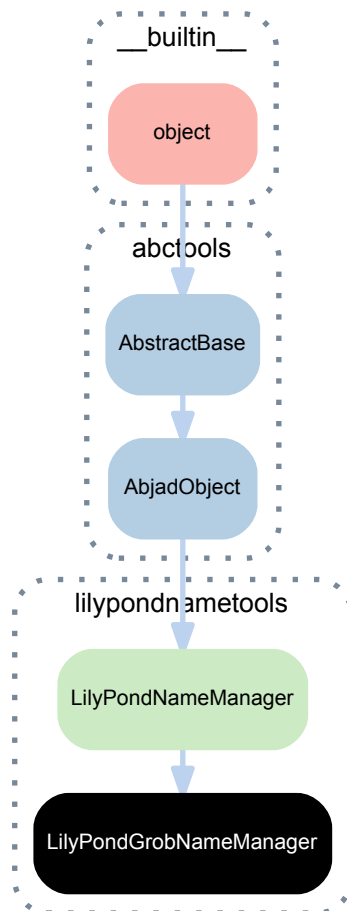
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

36.1.2 lilypondnametools.LilyPondGrobNameManager



class `lilypondnametools.LilyPondGrobNameManager`
 LilyPond grob name manager.

Bases

- `lilypondnametools.LilyPondNameManager`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Special methods

`(LilyPondNameManager).__eq__(arg)`

Is 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 Abjad 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.

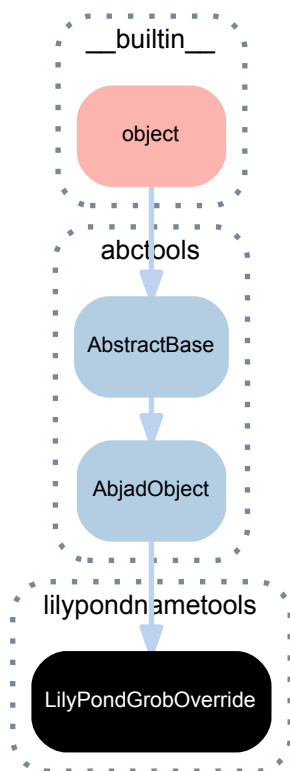
`(LilyPondNameManager).__hash__()`
 Hashes LilyPond name manager.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 Returns integer.

`(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.

36.1.3 lilypondnametools.LilyPondGrobOverride



```
class lilypondnametools.LilyPondGrobOverride(context_name=None,
                                             grob_name='NoteHead', is_once=None,
                                             is_revert=None, property_path='color',
                                             value='red')
```

A LilyPond grob override.

```
>>> override = lilypondnametools.LilyPondGrobOverride(
...     context_name='Staff',
...     grob_name='TextSpanner',
```

```

...     is_once=True,
...     property_path=(
...         'bound-details',
...         'left',
...         'text',
...     ),
...     value=Markup(r'\bold { over pressure }'),
... )

```

```

>>> print('\n'.join(override.override_format_pieces))
\once \override Staff.TextSpanner.bound-details.left.text = \markup {
  \bold
  {
    over
    pressure
  }
}

```

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`LilyPondGrobOverride.context_name`

Optional LilyPond grob override context name.

Returns string or none.

`LilyPondGrobOverride.grob_name`

LilyPond grob override grob name.

Returns string.

`LilyPondGrobOverride.is_once`

Is true if grob override is to be applied only once. Otherwise false.

Returns boolean or none.

`LilyPondGrobOverride.is_revert`

Is true if grob override is a grob revert. Otherwise false.

Returns boolean or none.

`LilyPondGrobOverride.override_format_pieces`

Gets LilyPond grob override override format pieces.

Returns tuple of strings.

`LilyPondGrobOverride.property_path`

LilyPond grob override property path.

Returns tuple of strings.

`LilyPondGrobOverride.revert_format_pieces`

Gets LilyPond grob override revert format pieces.

Returns tuple of strings.

`LilyPondGrobOverride.value`

Value of LilyPond grob override.

Returns arbitrary object.

Special methods

`LilyPondGrobOverride.__eq__(expr)`

Is true when *expr* is a LilyPond grob override with equivalent keyword values.

`(AbjadObject).__format__(format_specification='')`

Formats Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`LilyPondGrobOverride.__hash__()`

Hashes LilyPond grob override.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

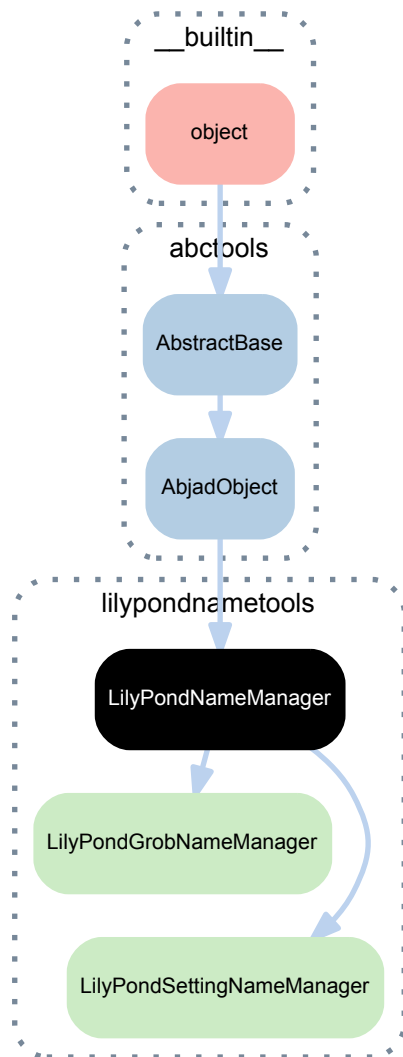
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

36.1.4 lilypondnametools.LilyPondNameManager



class `lilypondnametools.LilyPondNameManager`
 Base class from which LilyPond grob and setting managers inherit.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Special methods

`LilyPondNameManager.__eq__(arg)`

Is 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 Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`LilyPondNameManager.__hash__()`

Hashes LilyPond name manager.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

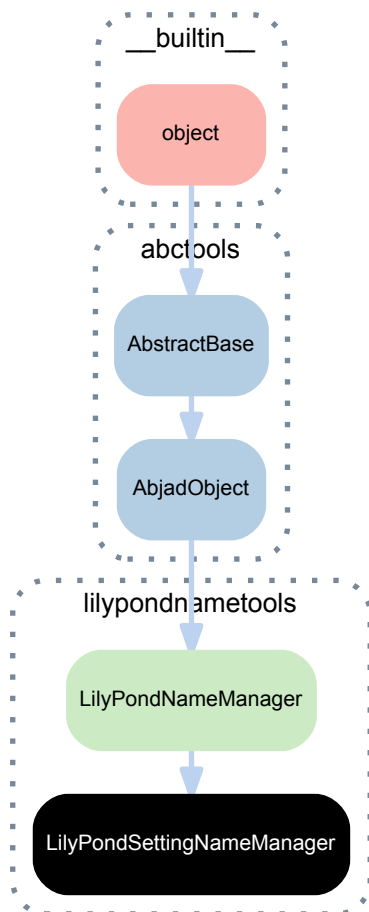
Returns boolean.

`LilyPondNameManager.__repr__()`

Gets interpreter representation of LilyPond name manager.

Returns string.

36.1.5 lilypondnametools.LilyPondSettingNameManager



class `lilypondnametools.LilyPondSettingNameManager`
 LilyPond setting name manager.

Bases

- `lilypondnametools.LilyPondNameManager`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`

- `__builtin__.object`

Special methods

`(LilyPondNameManager).__eq__(arg)`

Is 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 Abjad 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.

`(LilyPondNameManager).__hash__()`

Hashes LilyPond name manager.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

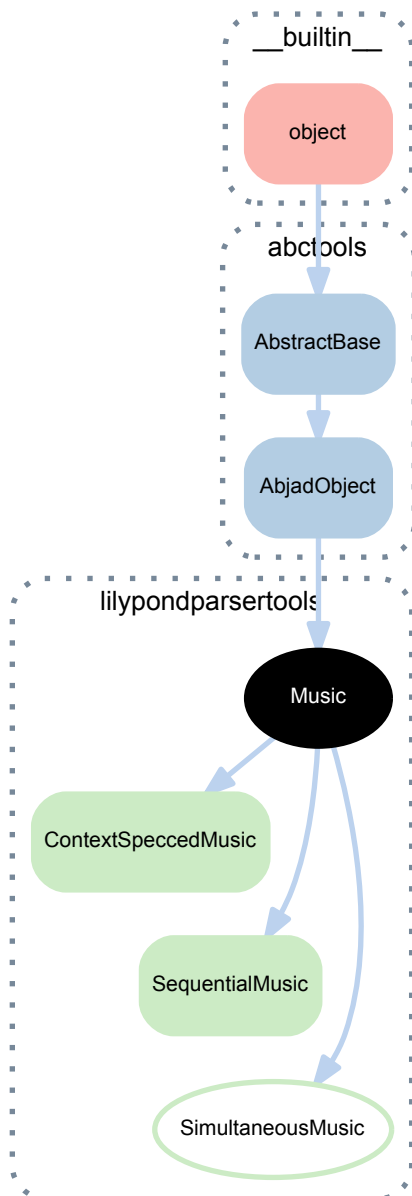
`(LilyPondNameManager).__repr__()`

Gets interpreter representation of LilyPond name manager.

Returns string.

37.1 Abstract classes

37.1.1 lilypondparsertools.Music



class `lilypondparsertools.Music` (*music=None*)
Abjad model of the LilyPond AST music node.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

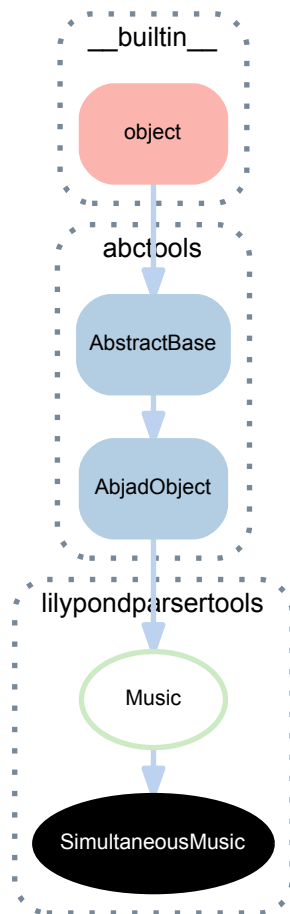
`(AbjadObject).__format__(format_specification='')`
Formats Abjad object.
Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
Returns string.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.
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.

37.1.2 lilypondparsertools.SimultaneousMusic



class `lilypondparsertools.SimultaneousMusic` (*music=None*)
 Abjad model of the LilyPond AST simultaneous music node.

Bases

- `lilypondparsertools.Music`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(AbjadObject). **__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(AbjadObject). **__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

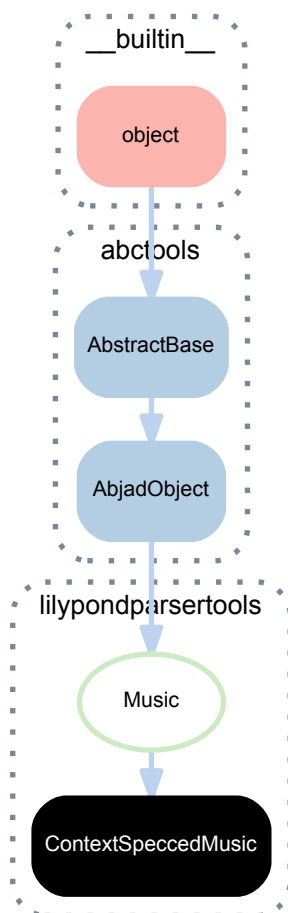
(AbjadObject). **__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

37.2 Concrete classes

37.2.1 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

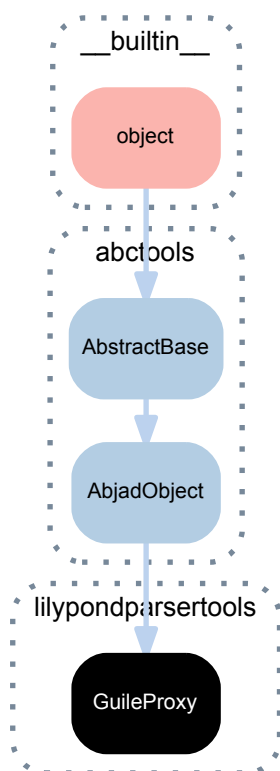
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

37.2.2 lilypondparsertools.GuileProxy



class `lilypondparsertools.GuileProxy` (*client=None*)

Emulates LilyPond music functions.

Used internally by LilyPondParser.

Not composer-safe.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.

(AbjadObject).**__hash__** ()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

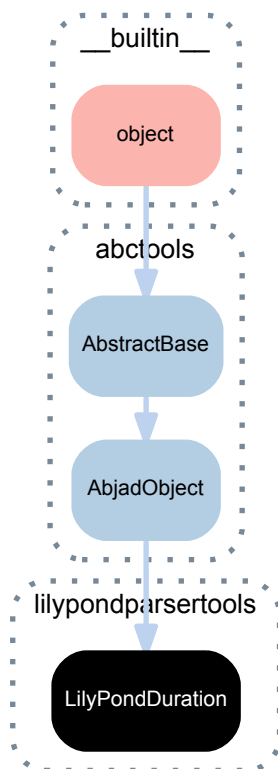
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

37.2.3 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

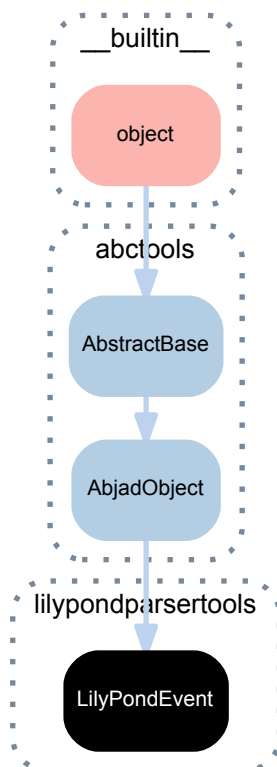
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

37.2.4 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`
- `abctools.AbjadObject.AbstractBase`

- `__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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

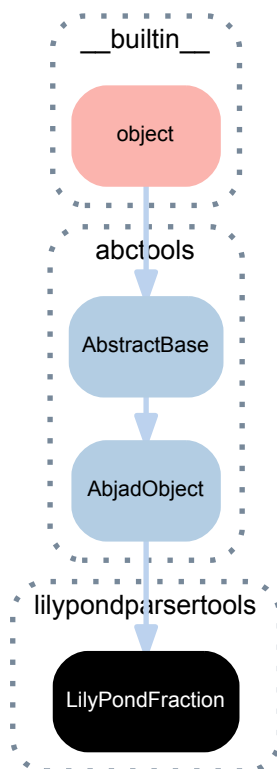
Returns boolean.

`LilyPondEvent.__repr__()`

Gets interpreter representation of LilyPond event.

Returns string.

37.2.5 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

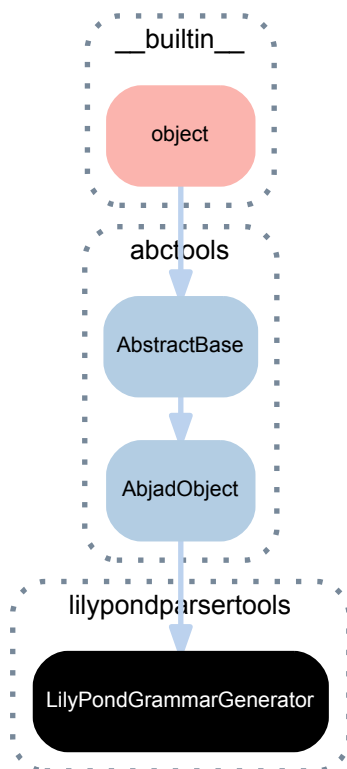
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

37.2.6 lilypondparsertools.LilyPondGrammarGenerator



class `lilypondparsertools.LilyPondGrammarGenerator`
 Generates a syntax skeleton from LilyPond grammar files.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(`AbjadObject`) `.__hash__`()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

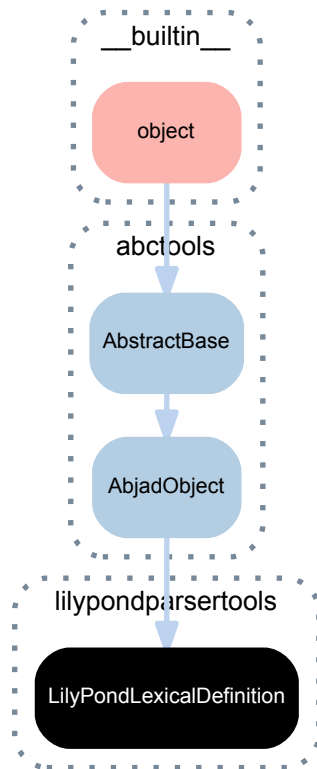
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

37.2.7 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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Methods

`LilyPondLexicalDefinition.push_signature` (*signature*, *t*)

`LilyPondLexicalDefinition.scan_bare_word` (*t*)

`LilyPondLexicalDefinition.scan_escaped_word` (*t*)

```

LilyPondLexicalDefinition.t_651_a(t)
  (((-?[0-9]+).[0-9]*)(-?.[0-9]+))

LilyPondLexicalDefinition.t_651_b(t)
  -[0-9]+

LilyPondLexicalDefinition.t_661(t)
  -.

LilyPondLexicalDefinition.t_666(t)
  [0-9]+

LilyPondLexicalDefinition.t_ANY_165(t)
  r

LilyPondLexicalDefinition.t_INITIAL_643(t)
  [a-zA-Z200-377](((a-zA-Z200-377)|_)[0-9])|)*

LilyPondLexicalDefinition.t_INITIAL_646(t)
  \a-zA-Z200-377](((a-zA-Z200-377)|_)[0-9])|)*

LilyPondLexicalDefinition.t_INITIAL_markup_notes_210(t)
  %{

LilyPondLexicalDefinition.t_INITIAL_markup_notes_214(t)
  %[^{nr}[^nr]*nr]

LilyPondLexicalDefinition.t_INITIAL_markup_notes_216(t)
  %[^{nr]

LilyPondLexicalDefinition.t_INITIAL_markup_notes_218(t)
  %nr]

LilyPondLexicalDefinition.t_INITIAL_markup_notes_220(t)
  %[^{nr}[^nr]*

LilyPondLexicalDefinition.t_INITIAL_markup_notes_222(t)
  [

  ]

LilyPondLexicalDefinition.t_INITIAL_markup_notes_227(t)
  “

LilyPondLexicalDefinition.t_INITIAL_markup_notes_353(t)
  #

LilyPondLexicalDefinition.t_INITIAL_notes_233(t)
  \version[ ntf]r]*

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)
    [\^#{\}\ttrf]+

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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

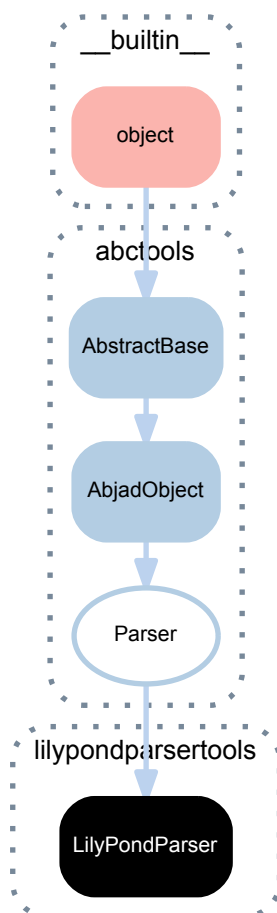
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

37.2.8 lilypondparsertools.LilyPondParser



class `lilypondparsertools.LilyPondParser` (*default_language='english', debug=False*)
 Parses a subset of LilyPond input syntax.

```
>>> parser = lilypondparsertools.LilyPondParser()
>>> string = r"\new Staff { c'4 ( d'8 e' fs'2) \fermata }"
>>> result = parser(string)
>>> 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')
>>> string = '{ c des e fis }'
>>> result = parser(string)
>>> 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, incl. 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 assigns (ie `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, \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 #)

Bases

- `abctools.Parser`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 }')
Container('c4 df4 e4 fs4')
```

```
>>> parser.default_language = 'nederlands'
>>> parser.default_language
'nederlands'
```

```
>>> parser('{ c des e fis }')
Container('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(contents=(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
Devnull
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_dynamics()`

Lists all dynamics recognized by LilyPond parser.

```
>>> for x in lilypondparsertools.LilyPondParser.list_known_dynamics():
...     print(x)
...
f
ff
fff
ffff
fffff
fp
fz
mf
mp
p
pp
ppp
pppp
ppppp
rfz
sf
sff
```

```
sfp
sfz
sp
spp
```

Returns tuple.

`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
ClefModifier
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
KievanLigature
LaissezVibrerTie
```

LaissezVibrerTieColumn
 LedgerLineSpanner
 LeftEdge
 LigatureBracket
 LyricExtender
 LyricHyphen
 LyricSpace
 LyricText
 MeasureCounter
 MeasureGrouping
 MelodyItem
 MensuralLigature
 MetronomeMark
 MultiMeasureRest
 MultiMeasureRestNumber
 MultiMeasureRestText
 NonMusicalPaperColumn
 NoteCollision
 NoteColumn
 NoteHead
 NoteName
 NoteSpacing
 OttavaBracket
 PaperColumn
 ParenthesesItem
 PercentRepeat
 PercentRepeatCounter
 PhrasingSlur
 PianoPedalBracket
 RehearsalMark
 RepeatSlash
 RepeatTie
 RepeatTieColumn
 Rest
 RestCollision
 Script
 ScriptColumn
 ScriptRow
 Slur
 SostenutoPedal
 SostenutoPedalLineSpanner
 SpacingSpanner
 SpanBar
 SpanBarStub
 StaffGrouper
 StaffSpacing
 StaffSymbol
 StanzaNumber
 Stem
 StemStub
 StemTremolo
 StringNumber
 StrokeFinger
 SustainPedal
 SustainPedalLineSpanner
 System
 SystemStartBar
 SystemStartBrace
 SystemStartBracket
 SystemStartSquare
 TabNoteHead
 TextScript
 TextSpanner
 Tie
 TieColumn
 TimeSignature
 TrillPitchAccidental
 TrillPitchGroup
 TrillPitchHead
 TrillSpanner
 TupletBracket
 TupletNumber
 UnaCordaPedal

```

UnaCordaPedalLineSpanner
VaticanaLigature
VerticalAlignment
VerticalAxisGroup
VoiceFollower
VoltaBracket
VoltaBracketSpanner

```

Returns tuple.

`LilyPondParser.list_known_languages()`

Lists all note-input languages recognized by LilyPond parser.

```

>>> 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-dashed-line
draw-dotted-line
draw-hline
draw-line
dynamic
ellipse
epsfile
eyeglasses
fermata
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-line
justify-string
large
larger
left-align
left-brace
left-column
line
lookup
lower
magnify
map-markup-commands
markalphabet
markletter
medium
musicglyph
natural
normal-size-sub
normal-size-super
normal-text
normalsize
note
note-by-number
null
number
on-the-fly
oval
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
score-lines
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 Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

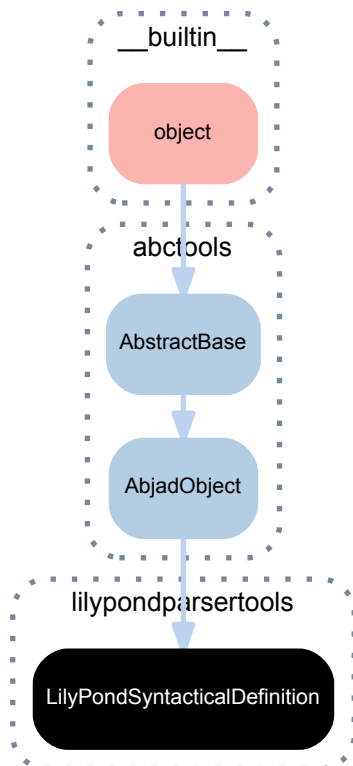
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

37.2.9 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`
- `abctools.AbjadObject.AbstractBase`
- `__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 : ‘\’

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__Chr123 (p)
    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_ch
    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 (p)
    event_chord : CHORD_REPETITION optional_notemode_duration post_events

LilyPondSyntacticalDefinition.p_event_chord__MULTI_MEASURE_REST__optional_notemode_duration (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__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_closed_keep du-
        ration_length

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_PITCH__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_keep
        pitch_also_in_chords

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_backup BACKUP

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' NUM-
        BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' UN-
        SIGNED

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep FRAC-
        TION

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep NUM-
        BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep REAL

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep UN-
        SIGNED

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep
        post_event_nofinger

LilyPondSyntacticalDefinition.p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu
    function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_keep embed-
        ded_scm_arg_closed

LilyPondSyntacticalDefinition.p_function_arglist_backup__function_arglist_backup__REPARS
    function_arglist_backup : function_arglist_backup REPARSE bare_number

LilyPondSyntacticalDefinition.p_function_arglist_backup__function_arglist_backup__REPARS
    function_arglist_backup : function_arglist_backup REPARSE embedded_scm_arg_closed

LilyPondSyntacticalDefinition.p_function_arglist_backup__function_arglist_backup__REPARS
    function_arglist_backup : function_arglist_backup REPARSE fraction

LilyPondSyntacticalDefinition.p_function_arglist_bare__EXPECT_DURATION__function_arglist
    function_arglist_bare : EXPECT_DURATION function_arglist_closed_optional duration_length

LilyPondSyntacticalDefinition.p_function_arglist_bare__EXPECT_NO_MORE_ARGS (p)
    function_arglist_bare : EXPECT_NO_MORE_ARGS

LilyPondSyntacticalDefinition.p_function_arglist_bare__EXPECT_OPTIONAL__EXPECT_DURATION__
    function_arglist_bare : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_skip DEFAULT

LilyPondSyntacticalDefinition.p_function_arglist_bare__EXPECT_OPTIONAL__EXPECT_PITCH__fu
    function_arglist_bare : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_skip DEFAULT

LilyPondSyntacticalDefinition.p_function_arglist_bare__EXPECT_OPTIONAL__EXPECT_SCM__func
    function_arglist_bare : EXPECT_OPTIONAL EXPECT_SCM function_arglist_skip DEFAULT

```

```

LilyPondSyntacticalDefinition.p_function_arglist_bare_EXPECT_PITCH_function_arglist_op
function_arglist_bare : EXPECT_PITCH function_arglist_optional pitch_also_in_chords

LilyPondSyntacticalDefinition.p_function_arglist_closed_function_arglist_closed_common
function_arglist_closed : function_arglist_closed_common

LilyPondSyntacticalDefinition.p_function_arglist_closed_function_arglist_nonbackup (p)
function_arglist_closed : function_arglist_nonbackup

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' NUM-
BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' UNSIGNED

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional bare_number

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional fraction

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional post_event_nofinger

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_EXPECT_SCM_function_arg
function_arglist_closed_common : EXPECT_SCM function_arglist_optional embedded_scm_arg_closed

LilyPondSyntacticalDefinition.p_function_arglist_closed_common_function_arglist_bare (p)
function_arglist_closed_common : function_arglist_bare

LilyPondSyntacticalDefinition.p_function_arglist_closed_keep_function_arglist_backup (p)
function_arglist_closed_keep : function_arglist_backup

LilyPondSyntacticalDefinition.p_function_arglist_closed_keep_function_arglist_closed_co
function_arglist_closed_keep : function_arglist_closed_common

LilyPondSyntacticalDefinition.p_function_arglist_closed_optional_EXPECT_OPTIONAL_EXPECT
function_arglist_closed_optional : EXPECT_OPTIONAL EXPECT_DURATION func-
tion_arglist_closed_optional

LilyPondSyntacticalDefinition.p_function_arglist_closed_optional_EXPECT_OPTIONAL_EXPECT
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_close
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_arglist_closed_optional__NUM-
    function_arglist_common_minus : EXPECT_SCM function_arglist_closed_optional '-' NUM-
    BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_common_minus__EXPECT_SCM__function_arglist_closed_optional__REAL
    function_arglist_common_minus : EXPECT_SCM function_arglist_closed_optional '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_common_minus__EXPECT_SCM__function_arglist_closed_optional__UNSIGNED
    function_arglist_common_minus : EXPECT_SCM function_arglist_closed_optional '-' UNSIGNED

LilyPondSyntacticalDefinition.p_function_arglist_common_minus__function_arglist_common_minus__REPARSE_bare_number
    function_arglist_common_minus : function_arglist_common_minus REPARSE bare_number

LilyPondSyntacticalDefinition.p_function_arglist_keep__function_arglist_backup (p)
    function_arglist_keep : function_arglist_backup

LilyPondSyntacticalDefinition.p_function_arglist_keep__function_arglist_common (p)
    function_arglist_keep : function_arglist_common

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_DURATION__function_arglist_closed_dura-
    tion_length
    function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_closed dura-
    tion_length

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_PITCH__function_arglist_closed_pitch_also_in_chords
    function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_PITCH function_arglist
    pitch_also_in_chords

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_embedded_scm_arg_closed
    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_closed__NUM-
    BER_IDENTIFIER
    function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' NUM-
    BER_IDENTIFIER

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed__REAL
    function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' REAL

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed__UNSIGNED
    function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' UN-
    SIGNED

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed__FRACTION
    function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed FRACTION

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_bare_number_closed
    function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed
    bare_number_closed

LilyPondSyntacticalDefinition.p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM__function_arglist_closed_post_event_nofinger
    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
    function_arglist_optional : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_optional

LilyPondSyntacticalDefinition.p_function_arglist_optional__EXPECT_OPTIONAL__EXPECT_PITCH__function_arglist_optional
    function_arglist_optional : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_optional

LilyPondSyntacticalDefinition.p_function_arglist_optional__function_arglist_backup__BACKUP
    function_arglist_optional : function_arglist_backup BACKUP

LilyPondSyntacticalDefinition.p_function_arglist_optional__function_arglist_keep (p)
    function_arglist_optional : function_arglist_keep %prec FUNCTION_ARGLIST

```

```

LilyPondSyntacticalDefinition.p_function_arglist_skip_EXPECT_OPTIONAL_EXPECT_DURATION__function_arglist_skip : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_function_arglist_skip_EXPECT_OPTIONAL_EXPECT_PITCH__function_arglist_skip : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_function_arglist_skip_EXPECT_OPTIONAL_EXPECT_SCM__function_arglist_skip : EXPECT_OPTIONAL EXPECT_SCM function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.p_function_arglist_skip__function_arglist_common (p)
    function_arglist_skip : function_arglist_common

LilyPondSyntacticalDefinition.p_gen_text_def__full_markup (p)
    gen_text_def : full_markup

LilyPondSyntacticalDefinition.p_gen_text_def__simple_string (p)
    gen_text_def : simple_string

LilyPondSyntacticalDefinition.p_grouped_music_list__sequential_music (p)
    grouped_music_list : sequential_music

LilyPondSyntacticalDefinition.p_grouped_music_list__simultaneous_music (p)
    grouped_music_list : simultaneous_music

LilyPondSyntacticalDefinition.p_identifier_init__context_def_spec_block (p)
    identifier_init : context_def_spec_block

LilyPondSyntacticalDefinition.p_identifier_init__context_modification (p)
    identifier_init : context_modification

LilyPondSyntacticalDefinition.p_identifier_init__embedded_scm (p)
    identifier_init : embedded_scm

LilyPondSyntacticalDefinition.p_identifier_init__full_markup (p)
    identifier_init : full_markup

LilyPondSyntacticalDefinition.p_identifier_init__full_markup_list (p)
    identifier_init : full_markup_list

LilyPondSyntacticalDefinition.p_identifier_init__music (p)
    identifier_init : music

LilyPondSyntacticalDefinition.p_identifier_init__number_expression (p)
    identifier_init : number_expression

LilyPondSyntacticalDefinition.p_identifier_init__output_def (p)
    identifier_init : output_def

LilyPondSyntacticalDefinition.p_identifier_init__post_event_nofinger (p)
    identifier_init : post_event_nofinger

LilyPondSyntacticalDefinition.p_identifier_init__score_block (p)
    identifier_init : score_block

LilyPondSyntacticalDefinition.p_identifier_init__string (p)
    identifier_init : string

LilyPondSyntacticalDefinition.p_lilypond__Empty (p)
    lilypond :

LilyPondSyntacticalDefinition.p_lilypond__lilypond__assignment (p)
    lilypond : lilypond assignment

LilyPondSyntacticalDefinition.p_lilypond__lilypond__error (p)
    lilypond : lilypond error

```

```

LilyPondSyntacticalDefinition.p_lilypond__lilypond__toplevel_expression (p)
    lilypond : lilypond toplevel_expression

LilyPondSyntacticalDefinition.p_lilypond_header__HEADER__Chr123__lilypond_header_body__Chr123 (p)
    lilypond_header : HEADER {' lilypond_header_body '}

LilyPondSyntacticalDefinition.p_lilypond_header_body__Empty (p)
    lilypond_header_body :

LilyPondSyntacticalDefinition.p_lilypond_header_body__lilypond_header_body__assignment (p)
    lilypond_header_body : lilypond_header_body assignment

LilyPondSyntacticalDefinition.p_markup__markup_head_1_list__simple_markup (p)
    markup : markup_head_1_list simple_markup

LilyPondSyntacticalDefinition.p_markup__simple_markup (p)
    markup : simple_markup

LilyPondSyntacticalDefinition.p_markup_braced_list__Chr123__markup_braced_list_body__Chr123 (p)
    markup_braced_list : {' markup_braced_list_body '}

LilyPondSyntacticalDefinition.p_markup_braced_list_body__Empty (p)
    markup_braced_list_body :

LilyPondSyntacticalDefinition.p_markup_braced_list_body__markup_braced_list_body__markup (p)
    markup_braced_list_body : markup_braced_list_body markup

LilyPondSyntacticalDefinition.p_markup_braced_list_body__markup_braced_list_body__markup_list (p)
    markup_braced_list_body : markup_braced_list_body markup_list

LilyPondSyntacticalDefinition.p_markup_command_basic_arguments__EXPECT_MARKUP_LIST__markup_command_list_arguments (p)
    markup_command_basic_arguments : EXPECT_MARKUP_LIST markup_command_list_arguments
    markup_list

LilyPondSyntacticalDefinition.p_markup_command_basic_arguments__EXPECT_NO_MORE_ARGS (p)
    markup_command_basic_arguments : EXPECT_NO_MORE_ARGS

LilyPondSyntacticalDefinition.p_markup_command_basic_arguments__EXPECT_SCM__markup_command_list_arguments__embedded_scm_closed (p)
    markup_command_basic_arguments : EXPECT_SCM markup_command_list_arguments embed-
    ded_scm_closed

LilyPondSyntacticalDefinition.p_markup_command_list__MARKUP_LIST_FUNCTION__markup_command_list_arguments (p)
    markup_command_list : MARKUP_LIST_FUNCTION markup_command_list_arguments

LilyPondSyntacticalDefinition.p_markup_command_list_arguments__EXPECT_MARKUP__markup_command_list_arguments_markup (p)
    markup_command_list_arguments : EXPECT_MARKUP markup_command_list_arguments markup

LilyPondSyntacticalDefinition.p_markup_command_list_arguments__markup_command_basic_arguments (p)
    markup_command_list_arguments : markup_command_basic_arguments

LilyPondSyntacticalDefinition.p_markup_composed_list__markup_head_1_list__markup_braced_list (p)
    markup_composed_list : markup_head_1_list markup_braced_list

LilyPondSyntacticalDefinition.p_markup_head_1_item__MARKUP_FUNCTION__EXPECT_MARKUP__markup_command_list_arguments (p)
    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_item (p)
    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__FRACTION (p)
    multiplied_duration : multiplied_duration '*' FRACTION

LilyPondSyntacticalDefinition.p_multiplied_duration__multiplied_duration__Chr42__bare_unsigned (p)
    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__embedded_scm_chord_body (p)
    music_function_chord_body_arglist : EXPECT_SCM music_function_chord_body_arglist embedded_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 (p)`
`property_operation : OVERRIDE simple_string property_path '=' scalar`

`LilyPondSyntacticalDefinition.p_property_operation__REVERT__simple_string__embedded_scm (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_closed (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 : 'l'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr43 (p)
    script_abbreviation : '+'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr45 (p)
    script_abbreviation : '-'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr46 (p)
    script_abbreviation : '.'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr94 (p)
    script_abbreviation : '^'

LilyPondSyntacticalDefinition.p_script_abbreviation__Chr95 (p)
    script_abbreviation : '_'

LilyPondSyntacticalDefinition.p_script_dir__Chr45 (p)
    script_dir : '-'

LilyPondSyntacticalDefinition.p_script_dir__Chr94 (p)
    script_dir : '^'

LilyPondSyntacticalDefinition.p_script_dir__Chr95 (p)
    script_dir : '_'

LilyPondSyntacticalDefinition.p_sequential_music__SEQUENTIAL__braced_music_list (p)
    sequential_music : SEQUENTIAL braced_music_list

LilyPondSyntacticalDefinition.p_sequential_music__braced_music_list (p)
    sequential_music : braced_music_list

LilyPondSyntacticalDefinition.p_simple_chord_elements__simple_element (p)
    simple_chord_elements : simple_element

LilyPondSyntacticalDefinition.p_simple_element__RESTNAME__optional_notemode_duration (p)
    simple_element : RESTNAME optional_notemode_duration

LilyPondSyntacticalDefinition.p_simple_element__pitch_exclamations_questions_octave_check optional_notemode_duration optional_rest (p)
    simple_element : pitch exclamations questions octave_check optional_notemode_duration optional_rest

LilyPondSyntacticalDefinition.p_simple_markup__MARKUP_FUNCTION__markup_command_basic_arguments (p)
    simple_markup : MARKUP_FUNCTION markup_command_basic_arguments

LilyPondSyntacticalDefinition.p_simple_markup__MARKUP_IDENTIFIER (p)
    simple_markup : MARKUP_IDENTIFIER

LilyPondSyntacticalDefinition.p_simple_markup__SCORE__Chr123__score_body__Chr125 (p)
    simple_markup : SCORE '{ score_body }'

LilyPondSyntacticalDefinition.p_simple_markup__STRING (p)
    simple_markup : STRING

LilyPondSyntacticalDefinition.p_simple_markup__STRING_IDENTIFIER (p)
    simple_markup : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.p_simple_markup__markup_scm__MARKUP_IDENTIFIER (p)
    simple_markup : markup_scm MARKUP_IDENTIFIER

LilyPondSyntacticalDefinition.p_simple_music__context_change (p)
    simple_music : context_change

LilyPondSyntacticalDefinition.p_simple_music__event_chord (p)
    simple_music : event_chord

```

```

LilyPondSyntacticalDefinition.p_simple_music__music_property_def (p)
    simple_music : music_property_def

LilyPondSyntacticalDefinition.p_simple_music_property_def__OVERRIDE__context_prop_spec__en
    simple_music_property_def : OVERRIDE context_prop_spec property_path '=' scalar

LilyPondSyntacticalDefinition.p_simple_music_property_def__REVERT__context_prop_spec__en
    simple_music_property_def : REVERT context_prop_spec embedded_scm

LilyPondSyntacticalDefinition.p_simple_music_property_def__SET__context_prop_spec__Chr61
    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__DOUB
    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 (p)
    tempo_event : TEMPO scalar_closed steno_duration ‘=’ tempo_range

LilyPondSyntacticalDefinition.p_tempo_event__TEMPO__steno_duration__Chr61__tempo_range (p)
    tempo_event : TEMPO steno_duration ‘=’ tempo_range

LilyPondSyntacticalDefinition.p_tempo_range__bare_unsigned (p)
    tempo_range : bare_unsigned

LilyPondSyntacticalDefinition.p_tempo_range__bare_unsigned__Chr45__bare_unsigned (p)
    tempo_range : bare_unsigned ‘-’ bare_unsigned

LilyPondSyntacticalDefinition.p_toplevel_expression__composite_music (p)
    toplevel_expression : composite_music

LilyPondSyntacticalDefinition.p_toplevel_expression__full_markup (p)
    toplevel_expression : full_markup

LilyPondSyntacticalDefinition.p_toplevel_expression__full_markup_list (p)
    toplevel_expression : full_markup_list

LilyPondSyntacticalDefinition.p_toplevel_expression__lilypond_header (p)
    toplevel_expression : lilypond_header

LilyPondSyntacticalDefinition.p_toplevel_expression__output_def (p)
    toplevel_expression : output_def

LilyPondSyntacticalDefinition.p_toplevel_expression__score_block (p)
    toplevel_expression : score_block

LilyPondSyntacticalDefinition.p_tremolo_type__Chr58 (p)
    tremolo_type : ‘:’

LilyPondSyntacticalDefinition.p_tremolo_type__Chr58__bare_unsigned (p)
    tremolo_type : ‘:’ bare_unsigned

```

Special methods

```

(AbjadObject).__eq__(expr)
    Is true when ID of expr equals ID of Abjad object. Otherwise false.

    Returns boolean.

(AbjadObject).__format__(format_specification='')
    Formats Abjad object.

    Set format_specification to ‘’ or ‘storage’. Interprets ‘’ equal to ‘storage’.

    Returns string.

(AbjadObject).__hash__()
    Hashes Abjad object.

    Required to be explicitly re-defined on Python 3 if __eq__ changes.

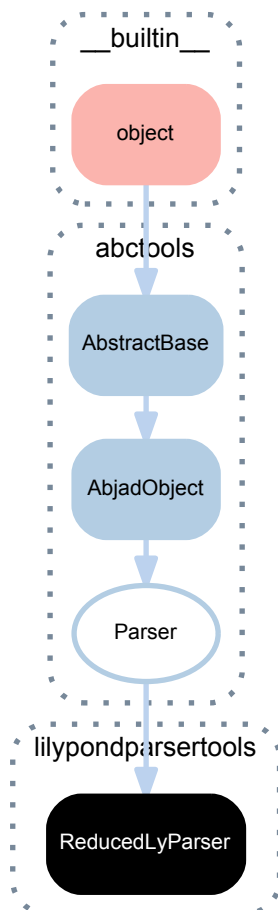
    Returns integer.

```

(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.

37.2.10 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))
\tweak #'edge-height #'(0.7 . 0)
\times 2/3 {
    c'4
    c'4
    \tweak #'text #tuplet-number::calc-fraction-text
    \tweak #'edge-height #'(0.7 . 0)
    \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`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`ReducedLyParser.debug`

Gets debug boolean of reduced ly parser.

Returns boolean.

`(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*)

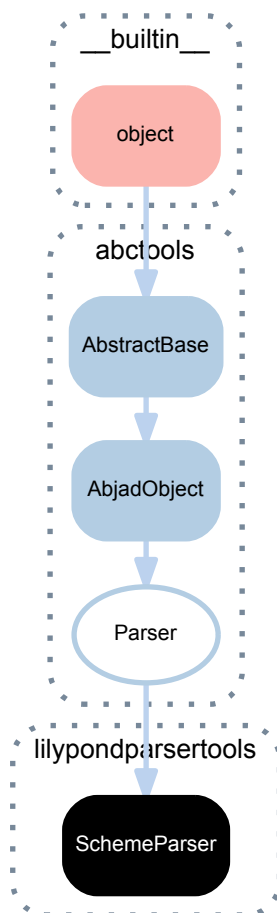
```


`ReducedLyParser.t_INTEGER_N(t)`
`(-[1-9]d*)`
`ReducedLyParser.t_INTEGER_P(t)`
`([1-9]d*)`
`ReducedLyParser.t_PITCHNAME(t)`
`[a-g](fflsslflslqtqlqlqlf)?`
`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 Abjad object.
 Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.
 Returns string.
`(AbjadObject).__hash__()`
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if `__eq__` changes.
 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.

37.2.11 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`
- `abctools.AbjadObject.AbstractBase`
- `__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.

(ObjadObject).__eq__(expr)
    Is true when ID of expr equals ID of Objad object. Otherwise false.
    Returns boolean.

(ObjadObject).__format__(format_specification='')
    Formats Objad object.
    Set format_specification to '' or 'storage'. Interprets '' equal to 'storage'.
    Returns string.

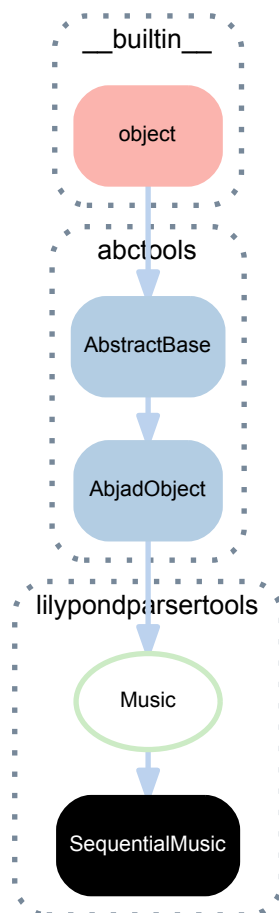
(ObjadObject).__hash__()
    Hashes Objad object.
    Required to be explicitly re-defined on Python 3 if __eq__ changes.
    Returns integer.

(ObjadObject).__ne__(expr)
    Is true when Objad object does not equal expr. Otherwise false.
    Returns boolean.

(ObjadObject).__repr__()
    Gets interpreter representation of Objad object.
    Returns string.

```

37.2.12 lilypondparsertools.SequentialMusic



class `lilypondparsertools.SequentialMusic` (*music=None*)
Abjad model of the LilyPond AST sequential music node.

Bases

- `lilypondparsertools.Music`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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.

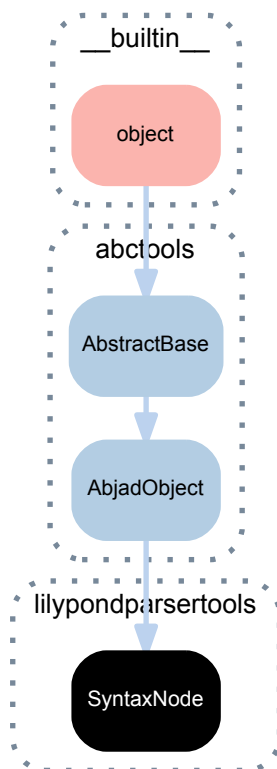
(AbjadObject).**__format__**(*format_specification*='')
 Formats Abjad object.
 Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.
 Returns string.

(AbjadObject).**__hash__**()
 Hashes Abjad object.
 Required to be explicitly re-defined on Python 3 if **__eq__** changes.
 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.

37.2.13 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`
- `abctools.AbjadObject.AbstractBase`

- `__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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`SyntaxNode.__getitem__(item)`

Gets *item* from syntax node.

Returns item.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`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.

37.3 Functions

37.3.1 lilypondparsertools.parse_reduced_ly_syntax

`lilypondparsertools.parse_reduced_ly_syntax(string)`

Parse the reduced LilyPond rhythmic syntax:

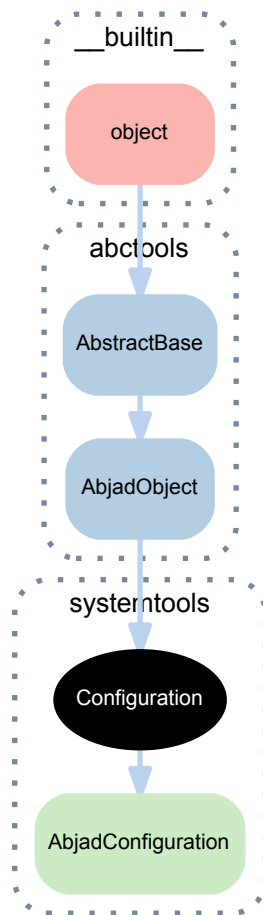
```
>>> string = '4 -4. 8.. 5/3 { } 4'
>>> result = lilypondparsertools.parse_reduced_ly_syntax(string)
```

```
>>> for x in result:
...     x
...
Note("c'4")
Rest('r4.')
Note("c'8..")
Tuplet(Multiplier(5, 3), '')
Note("c'4")
```


Returns list.

38.1 Abstract classes

38.1.1 systemtools.Configuration



class `systemtools.Configuration`
A configuration object.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`Configuration.configuration_directory`
Gets configuration directory.

Returns string.

`Configuration.configuration_file_name`
Gets configuration file name.

Returns string.

`Configuration.configuration_file_path`
Gets configuration file path.

Returns string.

`Configuration.home_directory`
Gets home directory.

Returns string.

Special methods

`Configuration.__delitem__(i)`
Deletes item *i* from configuration.

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 Abjad object.
Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`Configuration.__getitem__(i)`
Gets item *i* from configuration.

Returns none.

`(AbjadObject).__hash__()`
Hashes Abjad object.
Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`Configuration.__iter__()`
Iterates configuration settings.

Returns generator.

`Configuration.__len__()`
Gets the number of settings in configuration.

Returns nonnegative integer.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

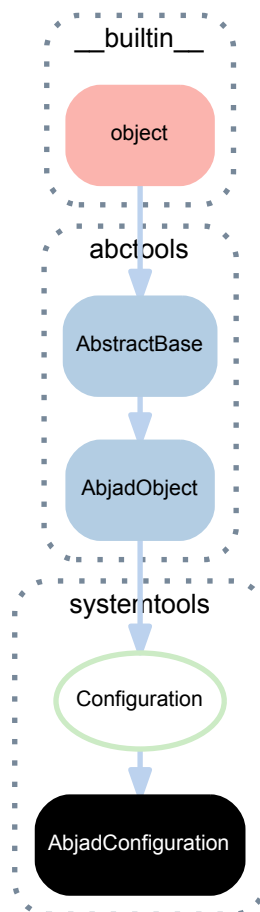
Returns boolean.

(AbjadObject).**__repr__**()
 Gets interpreter representation of Abjad object.
 Returns string.

Configuration.**__setitem__**(i, arg)
 Sets configuration item *i* to *arg*.
 Returns none.

38.2 Concrete classes

38.2.1 systemtools.AbjadConfiguration



class systemtools.**AbjadConfiguration**
 Abjad configuration.

```
>>> abjad_configuration = systemtools.AbjadConfiguration()
```

```
>>> abjad_configuration['accidental_spelling']
'mixed'
```

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_directory* key if it does not already exist.

AbjadConfiguration supports the mutable mapping interface and can be subscribed as a dictionary.

Bases

- `systemtools.Configuration`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`AbjadConfiguration.abjad_configuration_directory`
Abjad configuration directory.

Returns string.

`AbjadConfiguration.abjad_configuration_file_path`
Abjad configuration file path.

Returns string.

`AbjadConfiguration.abjad_directory`
Abjad directory.

Returns string.

`AbjadConfiguration.abjad_experimental_directory`
Abjad experimental directory.

Returns string.

`AbjadConfiguration.abjad_output_directory`
Abjad output directory.

Returns string.

`AbjadConfiguration.abjad_root_directory`
Abjad root directory.

Returns string.

`AbjadConfiguration.abjad_stylesheets_directory`
Abjad stylesheets directory.

Returns string.

`AbjadConfiguration.abjad_tools_directory`
Abjad tools directory.

Returns string.

`AbjadConfiguration.configuration_directory`
Configuration directory.

Returns string.

`AbjadConfiguration.configuration_file_name`
Configuration file name.

Returns string.

`(Configuration).configuration_file_path`

Gets configuration file path.

Returns string.

`(Configuration).home_directory`

Gets home directory.

Returns string.

`AbjadConfiguration.score_manager_directory`

Abjad score manager directory.

Returns string.

Class methods

`AbjadConfiguration.get_abjad_startup_string()`

Gets Abjad startup string.

```
>>> abjad_configuration.get_abjad_startup_string()
'Abjad 2.15 (development)'
```

Returns string.

`AbjadConfiguration.get_lilypond_minimum_version_string()`

Gets LilyPond minimum version string.

```
>>> abjad_configuration.get_lilypond_minimum_version_string()
'2.17.0'
```

Returns string.

Static methods

`AbjadConfiguration.get_abjad_version_string()`

Gets Abjad version string.

```
>>> abjad_configuration.get_abjad_version_string()
'2.15'
```

Returns string.

`AbjadConfiguration.get_lilypond_version_string()`

Gets LilyPond version string.

```
>>> abjad_configuration.get_lilypond_version_string()
'2.19.1'
```

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
```

Used by code generation functions.

Returns nonnegative integer.

`AbjadConfiguration.get_text_editor()`

Gets text editor.

```
>>> abjad_configuration.get_text_editor()
'vim'
```

Returns string.

`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')`

Sets default accidental spelling.

Sets 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")]
```

Sets 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")]
```

Sets 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")]
```

Defaults to 'mixed'.

Mixed test case must appear last here for doc tests to check correctly.

Returns none.

Special methods

`(Configuration).__delitem__(i)`

Deletes item *i* from configuration.

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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(Configuration).__getitem__(i)`

Gets item *i* from configuration.

Returns none.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(Configuration).**__iter__**()

Iterates configuration settings.

Returns generator.

(Configuration).**__len__**()

Gets the 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.

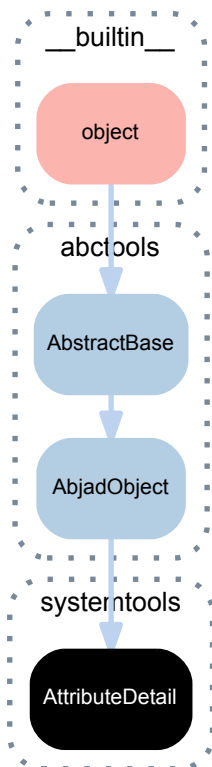
Returns string.

(Configuration).**__setitem__**(*i*, *arg*)

Sets configuration item *i* to *arg*.

Returns none.

38.2.2 systemtools.AttributeDetail



class systemtools.**AttributeDetail** (*display_string=None*, *editor=None*, *is_keyword=True*,
command=None, *name=None*)

Attribute detail.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`AttributeDetail.command`

Gets menu key of attribute detail.

Returns string.

`AttributeDetail.display_string`

Gets display string of attribute detail.

Returns string.

`AttributeDetail.editor`

Gets editor callable of attribute detail.

Returns callable.

`AttributeDetail.is_keyword`

Is true when attribute detail is keyword.

Returns boolean.

`AttributeDetail.name`

Gets name of attribute detail.

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 Abjad object.

Set *format_specification* to `''` or `'storage'`. Interprets `''` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

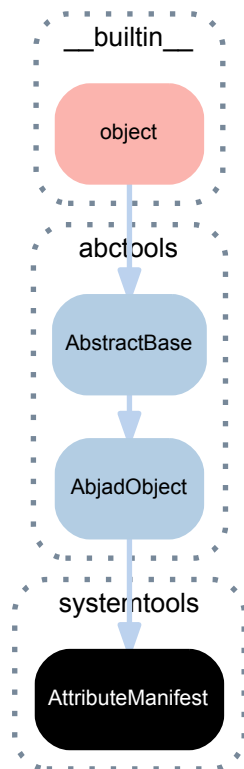
Returns boolean.

`AttributeDetail.__repr__()`

Gets interpreter representation of attribute detail.

Returns string.

38.2.3 systemtools.AttributeManifest



class `systemtools.AttributeManifest` (**attribute_details*)
 Target manifest.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`AttributeManifest.attribute_details`

Gets attribute details of target manifest.

Returns list.

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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`AttributeManifest.__getitem__(expr)`

Gets attribute detail from attribute manifest.

Returns attribute detail.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

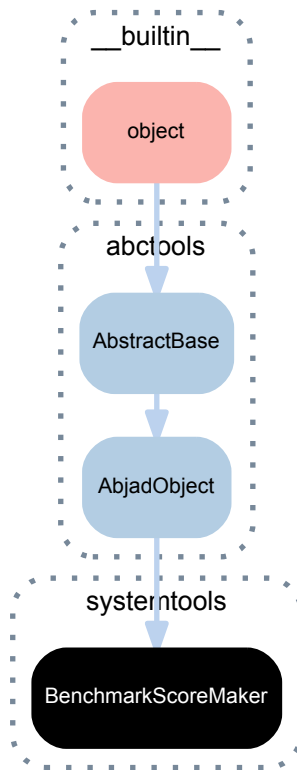
Returns boolean.

`AttributeManifest.__repr__()`

Gets interpreter representation of target manifest.

Returns string.

38.2.4 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`

- `abctools.AbjadObject.AbstractBase`
- `__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 r(9726) initialization:	235,126 function calls
2.12 (r9704) LilyPond format:	136,637 function calls
2.12 (r9710) LilyPond format:	82,730 function calls
2.12 (r9726) LilyPond format:	88,382 function calls

BenchmarkScoreMaker.make_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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject).**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

Returns integer.

(AbjadObject).**__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

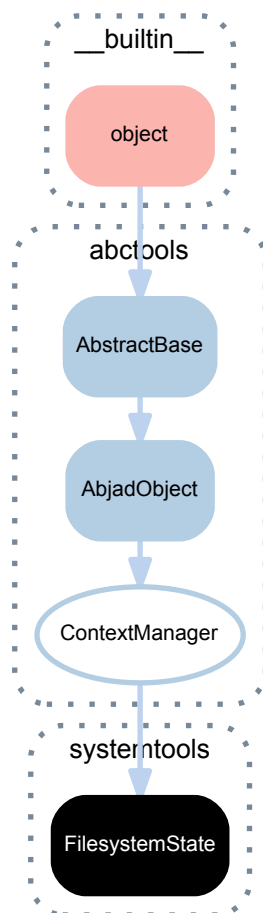
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

38.2.5 systemtools.FilesystemState



class `systemtools.FilesystemState` (*keep=None, remove=None*)
Filesystem state context manager.

Bases

- `abctools.ContextManager`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`FilesystemState.keep`
Gets asset paths to restore on exit.
Returns tuple.

`FilesystemState.remove`
Gets paths to remove on exit.
Returns tuple.

Special methods

`FileSystemState.__enter__()`

Backs up filesystem assets.

Returns none.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`FileSystemState.__exit__(exg_type, exc_value, traceback)`

Restores filesystem assets and removes backups; also removes paths in remove list.

Returns none.

`(AbjadObject).__format__(format_specification='')`

Formats Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

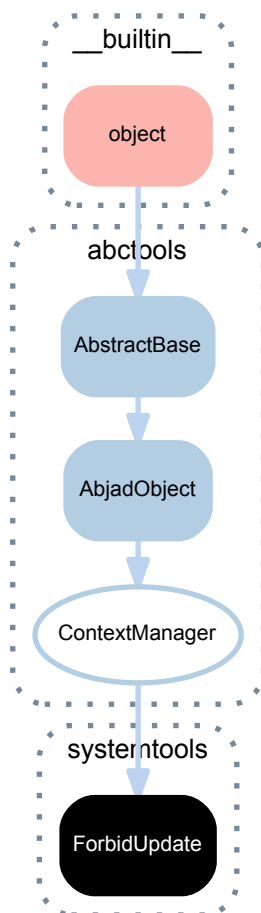
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

38.2.6 systemtools.ForbidUpdate



class `systemtools.ForbidUpdate` (*component=None*)
 A context manager for forbidding score updates.

```

>>> staff = Staff("c'32 d'2.. ~ d'16 e'32")
>>> with systemtools.ForbidUpdate(component=staff):
...     for x in staff[:]:
...         mutate(x).replace(Chord(x))
...
  
```

Bases

- `abctools.ContextManager`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`ForbidUpdate.component`
 Component of context manager.
 Return component.

Special methods

`ForbidUpdate.__enter__()`

Enters context manager.

Returns context manager.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`ForbidUpdate.__exit__(exc_type, exc_value, traceback)`

Exist context manager.

Returns none.

`(AbjadObject).__format__(format_specification='')`

Formats Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

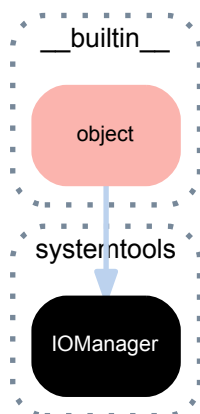
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

38.2.7 systemtools.IOManager



class `systemtools.IOManager`
 Manages Abjad IO.

Bases

- `__builtin__.object`

Static methods

`IOManager.clear_terminal()`

Clears terminal.

Runs `clear` if OS is POSIX-compliant (UNIX / Linux / MacOS).

Runs `cls` if OS is not POSIX-compliant (Windows).

Returns none.

`IOManager.count_function_calls(expr, global_context=None, local_context=None, fixed_point=True)`

Counts function calls required to execute *expr*.

Wraps `IOManager.profile_expr(expr)`.

Returns nonnegative integer.

`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=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* 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=None)`

Gets next output file name with *file_extension* in *output_directory*.

```
>>> systemtools.IOManager.get_next_output_file_name()
'6223.ly'
```

Gets next output file name with *file_extension* in Abjad output directory when *output_directory* is none.

Returns string.

`IOManager.make_subprocess(command)`

Makes Popen instance.

```
>>> command = 'echo "hellow world"'
>>> systemtools.IOManager.make_subprocess(command)
<subprocess.Popen object at 0x...>
```

Defined equal to

```
process = subprocess.Popen(
    command,
    shell=True,
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT,
)
```

Redirects `stderr` to `stdout`.

`IOManager.open_file(file_path, application=None, line_number=None)`

Opens *file_path*.

Uses *application* when *application* is not none.

Uses Abjad configuration file *text_editor* when *application* is none.

Takes best guess at operating system-specific file opener when both *application* and Abjad configuration file *text_editor* are none.

Respects *line_number* when *file_path* can be opened with text editor.

Returns none.

`IOManager.open_last_log()`

Opens LilyPond log file in operating system-specific text editor.

```
>>> systemtools.IOManager.open_last_log()
```

```
GNU LilyPond 2.19.2
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.open_last_ly(target=-1)`

Opens last LilyPond output file produced by Abjad.

Opens the last LilyPond output file:

```
>>> systemtools.IOManager.open_last_ly()
```

```
% 2014-02-12 14:29

\version "2.19.2"
\language "english"

\header {
  tagline = \markup {}
}

\layout {}

\paper {}

\score {
  {
    c'4
  }
}
```

Uses operating-specific text editor.

Set *target*=-2 to open the next-to-last LilyPond output file produced by Abjad, and so on.

Returns none.

`IOManager.open_last_pdf(target=-1)`

Opens last PDF generated by Abjad.

Abjad writes PDFs to the `~/ .abjad/output` directory by default.

You may change this by setting the `abjad_output_directory` variable in the `config.py` file.

Set *target*=-2 to open the next-to-last PDF generated by Abjad.

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("c8 c8 c8 c8 c8 c8 c8 c8")'
>>> 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      0.001      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_path, candidacy=False, flags=None, lily-pond_path=None)`

Runs LilyPond on *lilypond_file_path*.

Returns none.

`IOManager.save_last_ly_as(file_path)`

Saves last LilyPond file created by 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 by 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*.

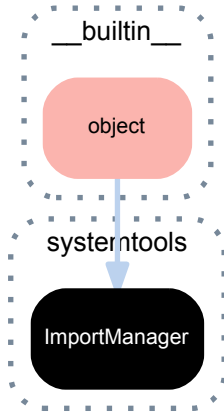
```
>>> command = 'echo "hello world"'
>>> IOManager.spawn_subprocess(command)
hello world
```

The function is basically a reimplementation of the deprecated `os.system()` using Python's `subprocess` module.

Redirects `stderr` to `stdout`.

Returns integer exit code.

38.2.8 `systemtools.ImportManager`



class `systemtools.ImportManager`
Imports structured packages.

Bases

- `__builtin__.object`

Static methods

`ImportManager.import_material_packages` (*path*, *namespace*)
Imports public materials from *path* into *namespace*.

This is the custom function that all AbjadIDE-managed scores may use to import public materials on startup.

`ImportManager.import_nominative_modules` (*path*, *namespace*)
Imports nominative modules from *path* into *namespace*.

`ImportManager.import_public_names_from_path_into_namespace` (*path*, *namespace*,
delete_systemtools=True)

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*.

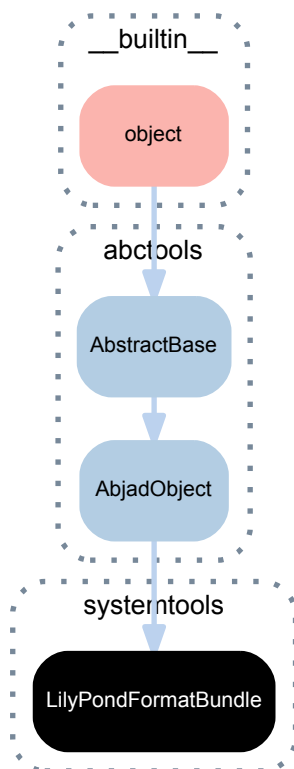
Does not inspect lower levels of *path*.

`ImportManager.import_structured_package` (*path*, *namespace*, *delete_systemtools=True*)
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.

38.2.9 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`
- `abctools.AbjadObject.AbstractBase`
- `__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()`

Alphabetizes format contributions in each slot.

Returns none.

`LilyPondFormatBundle.get(identifier)`

Gets *identifier*.

Returns format contributions object or list.

`LilyPondFormatBundle.make_immutable()`

Makes each slot immutable.

Returns none.

`LilyPondFormatBundle.update(format_bundle)`

Updates format bundle with all format contributions in *format_bundle*.

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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

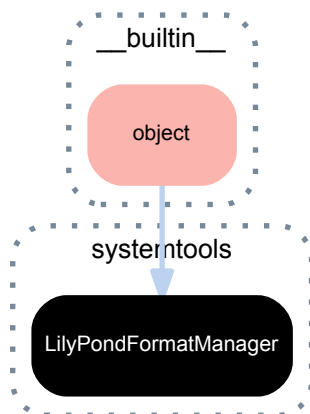
`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`(AbjadObject).__repr__()`
 Gets interpreter representation of Abjad object.
 Returns string.

38.2.10 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_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*)

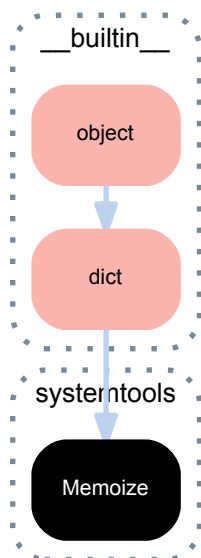
Reports spanner format contributions for every leaf to which spanner attaches.

```
>>> staff = Staff("c8 d e f")
>>> spanner = spannertools.Beam()
>>> attach(spanner, staff[:])
```

```
>>> manager = systemtools.LilyPondFormatManager
>>> print(manager.report_spanner_format_contributions(spanner))
c8 systemtools.LilyPondFormatBundle(
  right=LilyPondFormatBundle.SlotContributions(
    spanner_starts=['['],
  ),
)
d8 systemtools.LilyPondFormatBundle()
e8 systemtools.LilyPondFormatBundle()
f8 systemtools.LilyPondFormatBundle(
  right=LilyPondFormatBundle.SlotContributions(
    spanner_stops=['']],
  ),
)
```

Returns none or return string.

38.2.11 systemtools.Memoize



class `systemtools.Memoize` (*function=None*)
Memoize decorator.
Caches function return value.

Bases

- `__builtin__.dict`
- `__builtin__.object`

Methods

(dict). **clear** () → None. Remove all items from D.

(dict). **copy** () → a shallow copy of D

(dict). **get** (*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

(dict). **has_key** (*k*) → True if D has a key k, else False

(dict). **items** () → list of D's (key, value) pairs, as 2-tuples

(dict). **iteritems** () → an iterator over the (key, value) items of D

(dict). **iterkeys** () → an iterator over the keys of D

(dict). **itervalues** () → an iterator over the values of D

(dict). **keys** () → list of D's keys

(dict). **pop** (*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise `KeyError` is raised

(dict). **popitem** () → (k, v), remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if D is empty.

(dict). **setdefault** (*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

(dict). **update** ([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a `.keys()` method, does: for k in E: D[k] = E[k] If E present and lacks `.keys()` method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

(dict). **values** () → list of D's values

(dict).**viewitems**() → a set-like object providing a view on D's items

(dict).**viewkeys**() → a set-like object providing a view on D's keys

(dict).**viewvalues**() → an object providing a view on D's values

Special methods

Memoize.**__call__**(*args)

Calls decorator on *args*.

Calls function on *args* and caches if no cached value is found.

Returns cached value.

(dict).**__cmp__**(y) <==> *cmp*(x, y)

(dict).**__contains__**(k) → True if D has a key k, else False

(dict).**__delitem__**()

x.**__delitem__**(y) <==> del x[y]

(dict).**__eq__**()

x.**__eq__**(y) <==> x==y

(dict).**__ge__**()

x.**__ge__**(y) <==> x>=y

(dict).**__getitem__**()

x.**__getitem__**(y) <==> x[y]

(dict).**__gt__**()

x.**__gt__**(y) <==> x>y

(dict).**__iter__**() <==> *iter*(x)

(dict).**__le__**()

x.**__le__**(y) <==> x<=y

(dict).**__len__**() <==> *len*(x)

(dict).**__lt__**()

x.**__lt__**(y) <==> x<y

Memoize.**__missing__**(key)

Calls function on *key and caches.

Returns cached value.

(dict).**__ne__**()

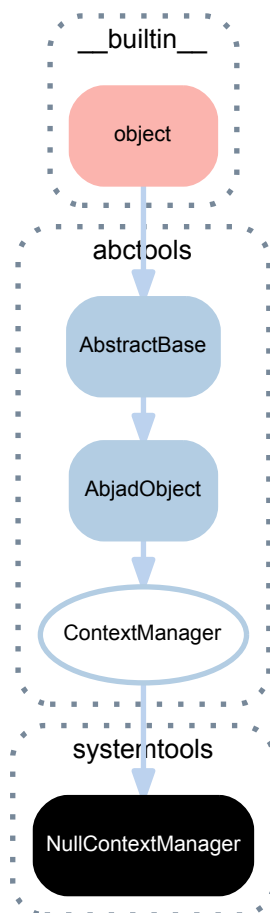
x.**__ne__**(y) <==> x!=y

(dict).**__repr__**() <==> *repr*(x)

(dict).**__setitem__**()

x.**__setitem__**(i, y) <==> x[i]=y

38.2.12 systemtools.NullContextManager



class `systemtools.NullContextManager`
A context manager that does nothing.

Bases

- `abctools.ContextManager`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Special methods

`NullContextManager.__enter__()`
Enters context manager and does nothing.

`(AbjadObject).__eq__(expr)`
Is true when ID of *expr* equals ID of Abjad object. Otherwise false.
Returns boolean.

`NullContextManager.__exit__(exc_type, exc_value, traceback)`
Exits context manager and does nothing.

`(AbjadObject).__format__(format_specification=''')`
Formats Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

(AbjadObject). **__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

(AbjadObject). **__ne__**(*expr*)

Is true when Abjad object does not equal *expr*. Otherwise false.

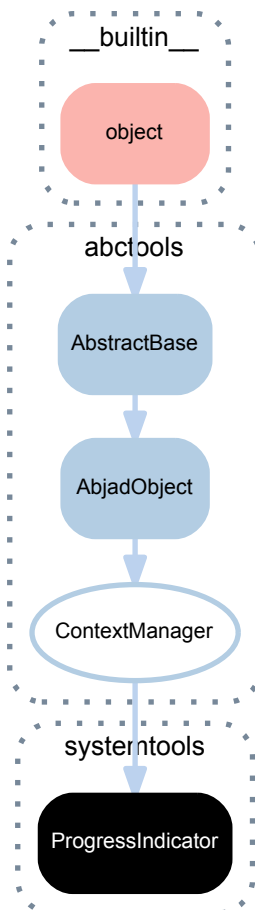
Returns boolean.

(AbjadObject). **__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

38.2.13 systemtools.ProgressIndicator



class systemtools.**ProgressIndicator** (*message*='')
A context manager for printing progress indications.

Bases

- `abctools.ContextManager`
- `abctools.AbjadObject`

- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`ProgressIndicator.message`

Gets message of progress indicator.

Returns string.

Methods

`ProgressIndicator.advance()`

Advances the progress indicator's progress count. Overwrites the current terminal line with the progress indicators message and new count.

Special methods

`ProgressIndicator.__enter__()`

Enters progress indicator.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`ProgressIndicator.__exit__(exc_type, exc_value, traceback)`

Exits progress indicator.

`(AbjadObject).__format__(format_specification='')`

Formats Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

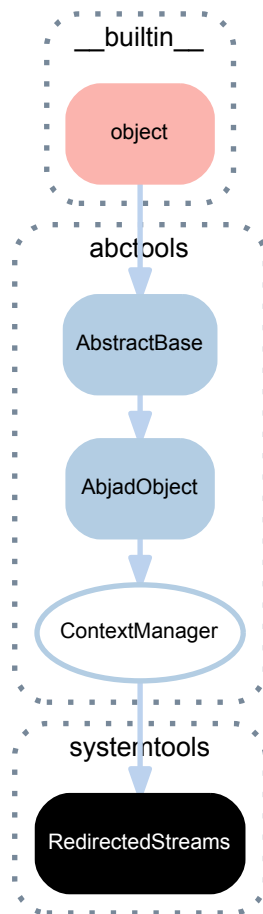
`ProgressIndicator.__repr__()`

Gets interpreter representation of context manager.

```
>>> context_manager = systemtools.ProgressIndicator()
>>> context_manager
<ProgressIndicator()>
```

Returns string.

38.2.14 systemtools.RedirectedStreams



class `systemtools.RedirectedStreams` (*stdout=None, stderr=None*)
 A context manager for capturing stdout and stderr output.

```

>>> try:
...     from StringIO import StringIO
... except ImportError:
...     from io import StringIO
...
>>> string_io = StringIO()
>>> with systemtools.RedirectedStreams(stdout=string_io):
...     print("hello, world!")
...
>>> result = string_io.getvalue()
>>> string_io.close()
>>> print(result)
hello, world!
  
```

Redirected streams context manager is immutable.

Bases

- `abctools.ContextManager`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`RedirectedStreams.stderr`
Gets stderr of context manager.

`RedirectedStreams.stdout`
Gets stdout of context manager.

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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__hash__()`
Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`
Is true when Abjad object does not equal *expr*. Otherwise false.

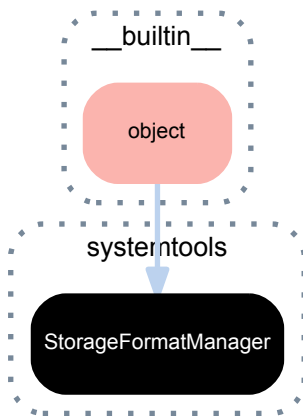
Returns boolean.

`RedirectedStreams.__repr__()`
Gets interpreter representation of context manager.

```
>>> context_manager = systemtools.RedirectedStreams()
>>> context_manager
<RedirectedStreams()>
```

Returns string.

38.2.15 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_hash_values(subject)`

Gets hash values for *subject*.

The hash values are a tuple of the type of *subject*, the values of its positional arguments, and the values of its keyword arguments, both sorted by argument name.

Return tuple.

`StorageFormatManager.get_indentation_strings(is_indented)`

Gets indentation strings.

`StorageFormatManager.get_input_argument_values(subject)`

Gets input argument values.

`StorageFormatManager.get_keyword_argument_dictionary(subject)`

Gets keyword argument dictionary.

`StorageFormatManager.get_keyword_argument_names(subject)`

Gets keyword argument names.

`StorageFormatManager.get_keyword_argument_values(subject)`

Gets keyword argument values.

`StorageFormatManager.get_positional_argument_dictionary(subject)`

Gets positional argument dictionary.

`StorageFormatManager.get_positional_argument_names (subject)`

Gets positional argument names.

`StorageFormatManager.get_positional_argument_values (subject)`

Gets positional argument values.

`StorageFormatManager.get_repr_format (subject)`

Gets interpreter representation format.

`StorageFormatManager.get_signature_keyword_argument_names (subject)`

Gets signature keyword argument names.

`StorageFormatManager.get_signature_positional_argument_names (subject)`

Gets signature positional argument names.

`StorageFormatManager.get_storage_format (subject)`

Gets storage format.

`StorageFormatManager.get_tools_package_name (subject)`

Gets tools-package name of *subject*.

```
>>> manager = systemtools.StorageFormatManager
>>> manager.get_tools_package_name(Note)
'scoretools'
```

`StorageFormatManager.get_tools_package_qualified_class_name (subject)`

Gets tools-package qualified class name of *subject*.

```
>>> manager = systemtools.StorageFormatManager
>>> manager.get_tools_package_qualified_class_name(Note)
'scoretools.Note'
```

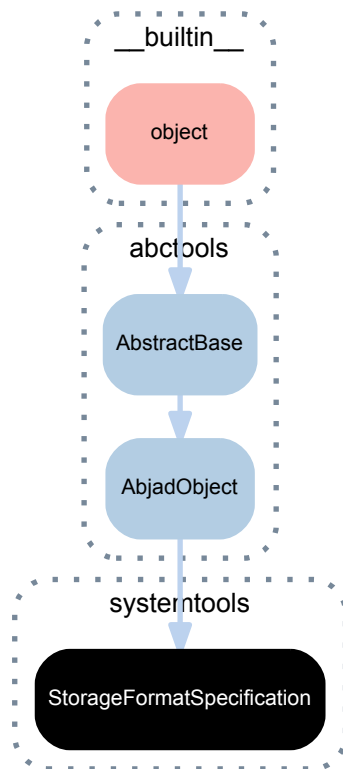
Returns string.

`StorageFormatManager.is_instance (subject)`

Is true when *subject* is instance. Otherwise false.

Returns boolean.

38.2.16 systemtools.StorageFormatSpecification



class `systemtools.StorageFormatSpecification` (*instance=None*, *body_text=None*,
is_bracketed=False, *is_indented=True*,
keyword_argument_callables=None,
keyword_argument_names=None,
keywords_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`
- `abctools.AbjadObject.AbstractBase`
- `__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_bracketed`

Is true when storage specification is bracketed. Otherwise false.

Returns boolean.

`StorageFormatSpecification.is_indented`

Is true when storage format is indented. Otherwise false.

Returns boolean.

`StorageFormatSpecification.keyword_argument_callables`

Keyword argument callables.

Returns tuple.

`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 Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

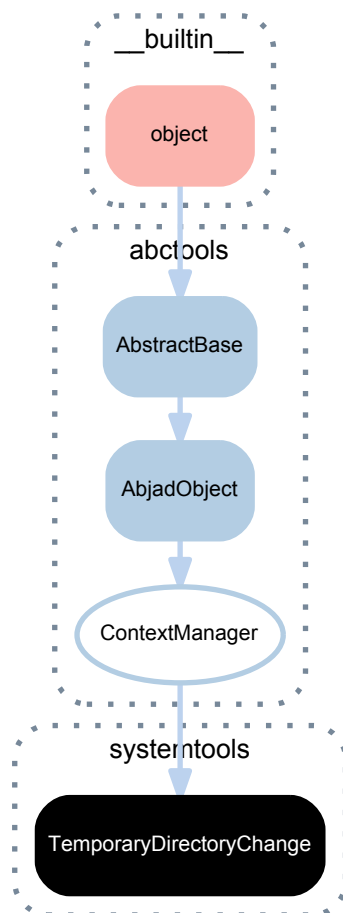
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

38.2.17 systemtools.TemporaryDirectoryChange



class `systemtools.TemporaryDirectoryChange` (*directory=None*)
 A context manager for temporarily changing the current working directory.

Bases

- `abctools.ContextManager`
- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__builtin__.object`

Read-only properties

`TemporaryDirectoryChange.directory`

Gets temporary directory of context manager.

Returns string.

Special methods

`TemporaryDirectoryChange.__enter__()`

Enters context manager and changes to *directory*.

`(AbjadObject).__eq__(expr)`

Is true when ID of *expr* equals ID of Abjad object. Otherwise false.

Returns boolean.

`TemporaryDirectoryChange.__exit__(exc_type, exc_value, traceback)`
 Exits context manager and returns to original working directory.

`(AbjadObject).__format__(format_specification='')`
 Formats Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`
 Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

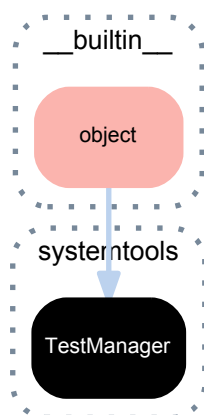
`(AbjadObject).__ne__(expr)`
 Is true when Abjad object does not equal *expr*. Otherwise false.

Returns boolean.

`TemporaryDirectoryChange.__repr__()`
 Gets interpreter representation of context manager.

Returns string.

38.2.18 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*.

Message newlines.

Returns boolean.

`TestManager.compare_files(path_1, path_2)`

Compares file *path_1* to file *path_2*.

For all file types:

```
* Performs line-by-line comparison
* Discards blank lines
```

For LilyPond files, additionally:

```
* Discards any LilyPond version statements
* Discards any lines beginning with ``%``
```

For PDFs, additionally discards lines that contain any of the following strings:

- /ID
- /CreationDate
- /ModDate
- xmp:CreateDate
- xmp:ModifyDate
- xapMM:DocumentID
- rdf:about

Discards first (binary) stream object in PDF; possibly first stream contains binary-encoded creator or times-tamp information that can vary from one creation of a PDF to another.

Returns true when files compare the same and false when files compare differently.

`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.

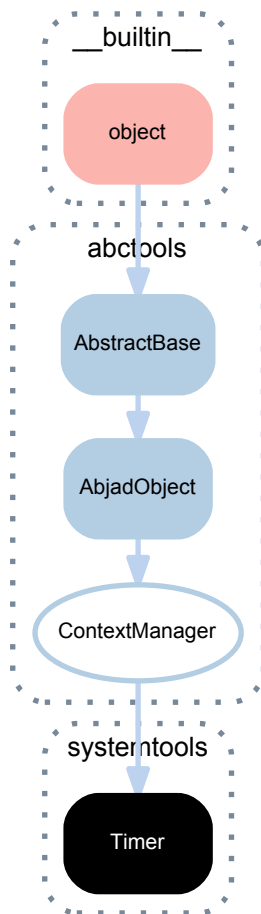
Returns list.

`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.

38.2.19 systemtools.Timer



class systemtools.Timer
 A timing context manager:

```

>>> timer = systemtools.Timer()
>>> with timer:
...     for _ in range(1000000):
...         x = 1 + 1
...
>>> timer.elapsed_time
0.092742919921875
  
```

The timer can also be accessed from within the *with* block:

```

>>> with systemtools.Timer() as timer:
...     for _ in range(5):
...         for _ in range(1000000):
...             x = 1 + 1
...             print(timer.elapsed_time)
...
0.101150989532
0.203935861588
0.304930925369
0.4057970047
0.50649189949
  
```

Timers can be reused between *with* blocks. They will reset their clock on entering any *with* block.

Bases

- `abctools.ContextManager`

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 context manager.

`(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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

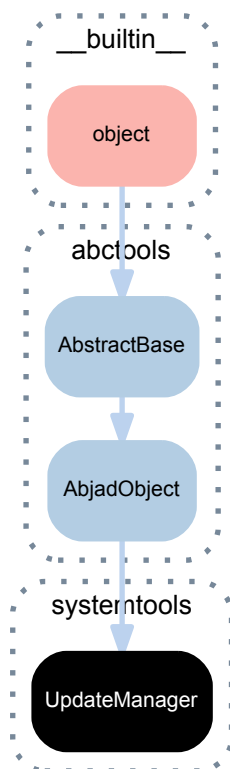
Returns boolean.

`(AbjadObject).__repr__()`

Gets interpreter representation of Abjad object.

Returns string.

38.2.20 systemtools.UpdateManager



class `systemtools.UpdateManager`

Update start offset, stop offsets and indicators everywhere in score.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to `'` or `'storage'`. Interprets `'` equal to `'storage'`.

Returns string.

`(AbjadObject).__hash__()`

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if `__eq__` changes.

Returns integer.

`(AbjadObject).__ne__(expr)`

Is true when Abjad object does not equal *expr*. Otherwise false.

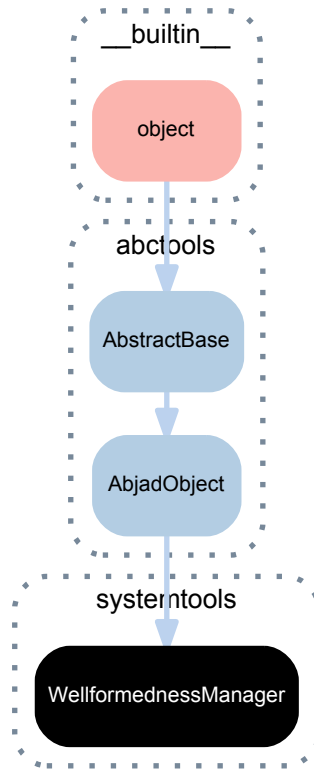
Returns boolean.

(AbjadObject).**__repr__**()

Gets interpreter representation of Abjad object.

Returns string.

38.2.21 systemtools.WellformednessManager



class systemtools.**WellformednessManager** (*expr=None, allow_empty_containers=True*)
Wellformedness manager.

Bases

- `abctools.AbjadObject`
- `abctools.AbjadObject.AbstractBase`
- `__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 Abjad object.

Set *format_specification* to '' or 'storage'. Interprets '' equal to 'storage'.

Returns string.

(AbjadObject) .**__hash__**()

Hashes Abjad object.

Required to be explicitly re-defined on Python 3 if **__eq__** changes.

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.

38.3 Functions

38.3.1 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.

38.3.2 systemtools.run_abjad

`systemtools.run_abjad()`

Runs Abjad.

Returns none.

A

- AbjadAPIGenerator (class in abjad.tools.documentationtools.AbjadAPIGenerator), 1444
- AbjadBookProcessor (class in abjad.tools.abjadbooktools.AbjadBookProcessor), 1377
- AbjadBookScript (class in abjad.tools.abjadbooktools.AbjadBookScript), 1379
- AbjadConfiguration (class in abjad.tools.systemtools.AbjadConfiguration), 1685
- AbjadObject (class in abjad.tools.abctools.AbjadObject), 1371
- AbjadValueObject (class in abjad.tools.abctools.AbjadValueObject), 1372
- AbjDevScript (class in abjad.tools.developerscripttools.AbjDevScript), 1394
- AbjGrepScript (class in abjad.tools.developerscripttools.AbjGrepScript), 1397
- Accidental (class in abjad.tools.pitchtools.Accidental), 523
- Accordion (class in abjad.tools.instrumenttools.Accordion), 223
- add_bell_music_to_score() (in module abjad.demos.part.add_bell_music_to_score), 1362
- add_string_music_to_score() (in module abjad.demos.part.add_string_music_to_score), 1362
- add_terminal_newlines() (in module abjad.tools.stringtools.add_terminal_newlines), 1187
- all_are_equal() (in module abjad.tools.mathtools.all_are_equal), 465
- all_are_integer_equivalent_exprs() (in module abjad.tools.mathtools.all_are_integer_equivalent_exprs), 465
- all_are_integer_equivalent_numbers() (in module abjad.tools.mathtools.all_are_integer_equivalent_numbers), 465
- all_are_nonnegative_integer_equivalent_numbers() (in module abjad.tools.mathtools.all_are_nonnegative_integer_equivalent_numbers), 466
- all_are_nonnegative_integer_powers_of_two() (in module abjad.tools.mathtools.all_are_nonnegative_integer_powers_of_two), 466
- all_are_nonnegative_integers() (in module abjad.tools.mathtools.all_are_nonnegative_integers), 466
- all_are_numbers() (in module abjad.tools.mathtools.all_are_numbers), 466
- all_are_pairs() (in module abjad.tools.mathtools.all_are_pairs), 467
- all_are_pairs_of_types() (in module abjad.tools.mathtools.all_are_pairs_of_types), 467
- all_are_positive_integer_equivalent_numbers() (in module abjad.tools.mathtools.all_are_positive_integer_equivalent_numbers), 467
- all_are_positive_integer_powers_of_two() (in module abjad.tools.mathtools.all_are_positive_integer_powers_of_two), 467
- all_are_positive_integers() (in module abjad.tools.mathtools.all_are_positive_integers), 468
- all_are_unequal() (in module abjad.tools.mathtools.all_are_unequal), 468
- AltoFlute (class in abjad.tools.instrumenttools.AltoFlute), 227
- AltoSaxophone (class in abjad.tools.instrumenttools.AltoSaxophone), 230
- AltoTrombone (class in abjad.tools.instrumenttools.AltoTrombone), 233
- AltoVoice (class in abjad.tools.instrumenttools.AltoVoice), 236
- AnnotatedTimespan (class in abjad.tools.timespantools.AnnotatedTimespan), 1211
- Annotation (class in abjad.tools.indicatortools.Annotation), 157
- append_spacer_skip_to_underfull_measure() (in module ab-

jad.tools.scoretools.append_spacer_skip_to_underfull_measures_in_expr(),
 1034
 append_spacer_skips_to_underfull_measures_in_expr() (in module abjad.tools.scoretools.append_spacer_skips_to_underfull_measures_in_expr(),
 1035
 apply_accidental_to_named_pitch() (in module abjad.tools.pitchtools.apply_accidental_to_named_pitch(),
 683
 apply_bowing_marks() (in module abjad.demos.part.apply_bowing_marks),
 1362
 apply_dynamics() (in module abjad.demos.part.apply_dynamics), 1362
 apply_expressive_marks() (in module abjad.demos.part.apply_expressive_marks),
 1362
 apply_final_bar_lines() (in module abjad.demos.part.apply_final_bar_lines),
 1363
 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(),
 1035
 apply_page_breaks() (in module abjad.demos.part.apply_page_breaks), 1363
 apply_rehearsal_marks() (in module abjad.demos.part.apply_rehearsal_marks),
 1363
 are_relatively_prime() (in module abjad.tools.mathtools.are_relatively_prime),
 468
 arg_to_bidirectional_direction_string() (in module abjad.tools.stringtools.arg_to_bidirectional_direction_string),
 1187
 arg_to_bidirectional_lilypond_symbol() (in module abjad.tools.stringtools.arg_to_bidirectional_lilypond_symbol),
 1187
 arg_to_tridirectional_direction_string() (in module abjad.tools.stringtools.arg_to_tridirectional_direction_string),
 1188
 arg_to_tridirectional_lilypond_symbol() (in module abjad.tools.stringtools.arg_to_tridirectional_lilypond_symbol),
 1188
 arg_to_tridirectional_ordinal_constant() (in module abjad.tools.stringtools.arg_to_tridirectional_ordinal_constant),
 1189
 arithmetic_mean() (in module abjad.tools.mathtools.arithmetic_mean),
 469
 Arpeggio (class in abjad.tools.indicator_tools.Arpeggio),
 159
 Articulation (class in abjad.tools.indicator_tools.Articulation), 160
 AssignabilityError (class in abjad.tools.exception_tools.AssignabilityError),
 1601
 attach() (in module abjad.tools.toplevel_tools.attach),
 1349
 AttackPointOptimizer (class in abjad.tools.quantization_tools.AttackPointOptimizer),
 697
 AttributeDetail (class in abjad.tools.system_tools.AttributeDetail),
 1689
 AttributeManifest (class in abjad.tools.system_tools.AttributeManifest),
 1691

B

BaritoneSaxophone (class in abjad.tools.instrument_tools.BaritoneSaxophone),
 239
 BaritoneVoice (class in abjad.tools.instrument_tools.BaritoneVoice),
 242
 BarLine (class in abjad.tools.indicator_tools.BarLine),
 163
 BaseResidueClass (class in abjad.tools.sieve_tools.BaseResidueClass),
 1110
 BassClarinet (class in abjad.tools.instrument_tools.BassClarinet),
 245
 BassFlute (class in abjad.tools.instrument_tools.BassFlute), 248
 Bassoon (class in abjad.tools.instrument_tools.Bassoon),
 260
 BassSaxophone (class in abjad.tools.instrument_tools.BassSaxophone),
 251
 BassTrombone (class in abjad.tools.instrument_tools.BassTrombone),
 254
 BassVoice (class in abjad.tools.instrument_tools.BassVoice), 257
 Beam (class in abjad.tools.spanner_tools.Beam), 1121
 BeamSpecifier (class in abjad.tools.rhythm_maker_tools.BeamSpecifier),
 803
 BeatwiseQSchema (class in abjad.tools.quantization_tools.BeatwiseQSchema),
 714
 BeatwiseQSchemaItem (class in abjad.tools.quantization_tools.BeatwiseQSchemaItem),
 719
 BeatwiseQTarget (class in abjad.tools.quantization_tools.BeatwiseQTarget),
 721
 BenchmarkScoreMaker (class in abjad.tools.system_tools.BenchmarkScoreMaker),
 1692
 BendAfter (class in abjad.tools.indicator_tools.BendAfter), 164
 binomial_coefficient() (in module abjad.tools.mathtools.binomial_coefficient),

- 469
- Block (class in abjad.tools.lilypondfiletools.Block), 413
- BoundedObject (class in abjad.tools.mathtools.BoundedObject), 447
- BowContactPoint (class in abjad.tools.indicatortools.BowContactPoint), 166
- BowPressure (class in abjad.tools.indicatortools.BowPressure), 168
- BuildApiScript (class in abjad.tools.developerscripttools.BuildApiScript), 1400
- BurnishSpecifier (class in abjad.tools.rhythmmakertools.BurnishSpecifier), 806
- ## C
- capitalize_start() (in module abjad.tools.stringtools.capitalize_start), 1189
- Cello (class in abjad.tools.instrumenttools.Cello), 263
- choose_mozart_measures() (in module abjad.demos.mozart.choose_mozart_measures), 1359
- Chord (class in abjad.tools.scoretools.Chord), 909
- ChordExtent (class in abjad.tools.tonalanalysistools.ChordExtent), 1313
- ChordInversion (class in abjad.tools.tonalanalysistools.ChordInversion), 1315
- ChordOmission (class in abjad.tools.tonalanalysistools.ChordOmission), 1317
- ChordQuality (class in abjad.tools.tonalanalysistools.ChordQuality), 1318
- ChordSuspension (class in abjad.tools.tonalanalysistools.ChordSuspension), 1319
- ClarinetInA (class in abjad.tools.instrumenttools.ClarinetInA), 266
- ClarinetInBFlat (class in abjad.tools.instrumenttools.ClarinetInBFlat), 269
- ClarinetInEFlat (class in abjad.tools.instrumenttools.ClarinetInEFlat), 272
- ClassCrawler (class in abjad.tools.documentationtools.ClassCrawler), 1446
- ClassDocumenter (class in abjad.tools.documentationtools.ClassDocumenter), 1447
- CleanScript (class in abjad.tools.developerscripttools.CleanScript), 1403
- Clef (class in abjad.tools.indicatortools.Clef), 169
- clef_and_staff_position_number_to_named_pitch() (in module abjad.tools.pitchtools.clef_and_staff_position_number_to_named), 683
- ClefInventory (class in abjad.tools.indicatortools.ClefInventory), 172
- Cluster (class in abjad.tools.scoretools.Cluster), 913
- CodeBlock (class in abjad.tools.abjadbooktools.CodeBlock), 1381
- CollapsingGraceHandler (class in abjad.tools.quantizationtools.CollapsingGraceHandler), 723
- color_chord_note_heads_in_expr_by_pitch_class_color_map() (in module abjad.tools.labeltools.color_chord_note_heads_in_expr_by_pitch), 391
- color_contents_of_container() (in module abjad.tools.labeltools.color_contents_of_container), 391
- color_leaf() (in module abjad.tools.labeltools.color_leaf), 392
- color_leaves_in_expr() (in module abjad.tools.labeltools.color_leaves_in_expr), 392
- color_measure() (in module abjad.tools.labeltools.color_measure), 393
- color_measures_with_non_power_of_two_denominators_in_expr() (in module abjad.tools.labeltools.color_measures_with_non_power_of_two_c), 393
- color_note_head_by_numbered_pitch_class_color_map() (in module abjad.tools.labeltools.color_note_head_by_numbered_pitch_class), 394
- combine_markup_commands() (in module abjad.tools.markuptools.combine_markup_commands), 444
- compare_images() (in module abjad.tools.documentationtools.compare_images), 1597
- ComplexBeam (class in abjad.tools.spannertools.ComplexBeam), 1124
- ComplexTrillSpanner (class in abjad.tools.spannertools.ComplexTrillSpanner), 1127
- Component (class in abjad.tools.scoretools.Component), 905
- CompoundInequality (class in abjad.tools.timespantools.CompoundInequality), 1232
- ConcatenatingGraceHandler (class in abjad.tools.quantizationtools.ConcatenatingGraceHandler), 724
- Configuration (class in abjad.tools.systemtools.Configuration), 1683

[configure_lilypond_file\(\)](#) (in module `abjad.demos.ferneyhough.configure_lilypond_file`), [1357](#)
[configure_lilypond_file\(\)](#) (in module `abjad.demos.part.configure_lilypond_file`), [1363](#)
[configure_score\(\)](#) (in module `abjad.demos.ferneyhough.configure_score`), [1357](#)
[configure_score\(\)](#) (in module `abjad.demos.part.configure_score`), [1363](#)
[Container](#) (class in `abjad.tools.scoretools.Container`), [919](#)
[contains_subsegment\(\)](#) (in module `abjad.tools.pitchtools.contains_subsegment`), [684](#)
[Context](#) (class in `abjad.tools.scoretools.Context`), [925](#)
[ContextBlock](#) (class in `abjad.tools.lilypondfiletools.ContextBlock`), [415](#)
[ContextManager](#) (class in `abjad.tools.abctools.ContextManager`), [1367](#)
[ContextMap](#) (class in `abjad.tools.datastructuretools.ContextMap`), [41](#)
[ContextSpeccedMusic](#) (class in `abjad.tools.lilypondparsertools.ContextSpeccedMusic`), [1628](#)
[ContiguousSelection](#) (class in `abjad.tools.selectiontools.ContiguousSelection`), [1051](#)
[Contrabass](#) (class in `abjad.tools.instrumenttools.Contrabass`), [275](#)
[ContrabassClarinet](#) (class in `abjad.tools.instrumenttools.ContrabassClarinet`), [278](#)
[ContrabassFlute](#) (class in `abjad.tools.instrumenttools.ContrabassFlute`), [281](#)
[Contrabassoon](#) (class in `abjad.tools.instrumenttools.Contrabassoon`), [287](#)
[ContrabassSaxophone](#) (class in `abjad.tools.instrumenttools.ContrabassSaxophone`), [284](#)
[CountLinewidthsScript](#) (class in `abjad.tools.developerscripttools.CountLinewidthsScript`), [1406](#)
[CountToolsScript](#) (class in `abjad.tools.developerscripttools.CountToolsScript`), [1409](#)
[create_pitch_contour_reservoir\(\)](#) (in module `abjad.demos.part.create_pitch_contour_reservoir`), [1363](#)
[Crescendo](#) (class in `abjad.tools.spannertools.Crescendo`), [1129](#)
[cumulative_products\(\)](#) (in module `abjad.tools.mathtools.cumulative_products`), [469](#)
[cumulative_signed_weights\(\)](#) (in module `abjad.tools.mathtools.cumulative_signed_weights`), [470](#)
[cumulative_sums\(\)](#) (in module `abjad.tools.mathtools.cumulative_sums`), [470](#)
[cumulative_sums_pairwise\(\)](#) (in module `abjad.tools.mathtools.cumulative_sums_pairwise`), [470](#)
[CyclicMatrix](#) (class in `abjad.tools.datastructuretools.CyclicMatrix`), [45](#)
[CyclicPayloadTree](#) (class in `abjad.tools.datastructuretools.CyclicPayloadTree`), [48](#)
[CyclicTuple](#) (class in `abjad.tools.datastructuretools.CyclicTuple`), [63](#)

D

[DateTimeToken](#) (class in `abjad.tools.lilypondfiletools.DateTimeToken`), [418](#)
[Decrescendo](#) (class in `abjad.tools.spannertools.Decrescendo`), [1133](#)
[delimit_words\(\)](#) (in module `abjad.tools.stringtools.delimit_words`), [1189](#)
[Descendants](#) (class in `abjad.tools.selectiontools.Descendants`), [1054](#)
[detach\(\)](#) (in module `abjad.tools.topleveltools.detach`), [1349](#)
[DeveloperScript](#) (class in `abjad.tools.developerscripttools.DeveloperScript`), [1389](#)
[difference_series\(\)](#) (in module `abjad.tools.mathtools.difference_series`), [470](#)
[DirectoryScript](#) (class in `abjad.tools.developerscripttools.DirectoryScript`), [1392](#)
[DiscardingGraceHandler](#) (class in `abjad.tools.quantizationtools.DiscardingGraceHandler`), [726](#)
[DistanceHeuristic](#) (class in `abjad.tools.quantizationtools.DistanceHeuristic`), [727](#)
[divide_number_by_ratio\(\)](#) (in module `abjad.tools.mathtools.divide_number_by_ratio`), [470](#)
[divisors\(\)](#) (in module `abjad.tools.mathtools.divisors`), [471](#)
[Documenter](#) (class in `abjad.tools.documentationtools.Documenter`), [1450](#)
[durate_pitch_contour_reservoir\(\)](#) (in module `abjad.demos.part.durate_pitch_contour_reservoir`),

- 1363
 DuratedComplexBeam (class in abjad.tools.spannertools.DuratedComplexBeam), 1137
 Duration (class in abjad.tools.durationtools.Duration), 121
 DurationSpellingSpecifier (class in abjad.tools.rhythmmakertools.DurationSpellingSpecifier), 811
 Dynamic (class in abjad.tools.indicatorertools.Dynamic), 177
- ## E
- edit_bass_voice() (in module abjad.demos.part.edit_bass_voice), 1363
 edit_cello_voice() (in module abjad.demos.part.edit_cello_voice), 1363
 edit_first_violin_voice() (in module abjad.demos.part.edit_first_violin_voice), 1363
 edit_second_violin_voice() (in module abjad.demos.part.edit_second_violin_voice), 1364
 edit_viola_voice() (in module abjad.demos.part.edit_viola_voice), 1364
 EnglishHorn (class in abjad.tools.instrumenttools.EnglishHorn), 290
 EvenRunRhythmMaker (class in abjad.tools.rhythmmakertools.EvenRunRhythmMaker), 814
 ExampleWrapper (class in abjad.tools.rhythmmakertools.ExampleWrapper), 818
 extend_measures_in_expr_and_apply_full_measure_tuplets() (in module abjad.tools.scoretools.extend_measures_in_expr_and_apply_full_measure_tuplets), 1035
 ExtraSpannerError (class in abjad.tools.exceptionertools.ExtraSpannerError), 1602
- ## F
- factors() (in module abjad.tools.mathtools.factors), 471
 FilesystemState (class in abjad.tools.systemtools.FilesystemState), 1696
 fill_measures_in_expr_with_full_measure_spacer_skips() (in module abjad.tools.scoretools.fill_measures_in_expr_with_full_measure_spacer_skips), 1036
 fill_measures_in_expr_with_minimal_number_of_notes() (in module abjad.tools.scoretools.fill_measures_in_expr_with_minimal_number_of_notes), 1036
 fill_measures_in_expr_with_repeated_notes() (in module abjad.tools.scoretools.fill_measures_in_expr_with_repeated_notes), 1036
 fill_measures_in_expr_with_time_signature_denominator_notes() (in module abjad.tools.scoretools.fill_measures_in_expr_with_time_signature_denominator_notes), 1036
 FixedDurationContainer (class in abjad.tools.scoretools.FixedDurationContainer), 932
 FixedDurationTuplet (class in abjad.tools.scoretools.FixedDurationTuplet), 938
 flatten_sequence() (in module abjad.tools.sequencetools.flatten_sequence), 1088
 Flute (class in abjad.tools.instrumenttools.Flute), 293
 ForbidUpdate (class in abjad.tools.systemtools.ForbidUpdate), 1698
 format_input_lines_as_doc_string() (in module abjad.tools.stringtools.format_input_lines_as_doc_string), 1190
 format_input_lines_as_regression_test() (in module abjad.tools.stringtools.format_input_lines_as_regression_test), 1190
 fraction_to_proper_fraction() (in module abjad.tools.mathtools.fraction_to_proper_fraction), 472
 FrenchHorn (class in abjad.tools.instrumenttools.FrenchHorn), 296
 FunctionCrawler (class in abjad.tools.documentationtools.FunctionCrawler), 1452
 FunctionDocumenter (class in abjad.tools.documentationtools.FunctionDocumenter), 1453
- ## G
- GalleryMaker (class in abjad.tools.rhythmmakertools.GalleryMaker), 819
 GeneralizedBeam (class in abjad.tools.spannertools.GeneralizedBeam), 1142
 get_developer_script_classes() (in module abjad.tools.developerscripttools.get_developer_script_classes), 1431
 get_measure_that_starts_with_container() (in module abjad.tools.scoretools.get_measure_that_starts_with_container), 1036
 get_measure_that_stops_with_container() (in module abjad.tools.scoretools.get_measure_that_stops_with_container), 1037
 get_named_pitch_from_pitch_carrier() (in module abjad.tools.pitchtools.get_named_pitch_from_pitch_carrier), 684
 get_repeated_measure_from_component() (in module ab-

[jad.tools.scoretools.get_next_measure_from_component\(\)](#), 473
[1037](#)
[get_numbered_pitch_class_from_pitch_carrier\(\)](#)
 (in module [ab-](#) [1199](#)
[jad.tools.pitchtools.get_numbered_pitch_class_from_pitch_carrier\(\)](#),
[684](#)
[get_one_indexed_measure_number_in_expr\(\)](#) [1201](#)
 (in module [ab-](#) [GuileProxy](#) (class in [ab-](#)
[jad.tools.scoretools.get_one_indexed_measure_number_in_expr\(\)](#),
[1037](#) [1630](#)
[get_previous_measure_from_component\(\)](#) [Guitar](#) (class in [abjad.tools.instrumenttools.Guitar](#)), 302
 (in module [ab-](#)
[jad.tools.scoretools.get_previous_measure_from_component\(\)](#),
[1038](#)
[get_shared_numeric_sign\(\)](#) (in module [ab-](#) [1147](#)
[jad.tools.mathtools.get_shared_numeric_sign\(\)](#), [Harp](#) (class in [abjad.tools.instrumenttools.Harp](#)), 305
[472](#) [Harpsichord](#) (class in [ab-](#)
[Glissando](#) (class in [abjad.tools.spannertools.Glissando](#)), [jad.tools.instrumenttools.Harpsichord](#),
[1145](#) [308](#)
[Glockenspiel](#) (class in [ab-](#) [Heuristic](#) (class in [ab-](#)
[jad.tools.instrumenttools.Glockenspiel](#)), [jad.tools.quantizationtools.Heuristic](#)), 701
[299](#)
[GraceContainer](#) (class in [ab-](#) [HiddenStaffSpanner](#) (class in [ab-](#)
[jad.tools.scoretools.GraceContainer](#)), 955 [jad.tools.spannertools.HiddenStaffSpanner](#)),
[1151](#)
[GraceHandler](#) (class in [ab-](#) [HorizontalBracketSpanner](#) (class in [ab-](#)
[jad.tools.quantizationtools.GraceHandler](#)), [jad.tools.spannertools.HorizontalBracketSpanner](#),
[699](#) [1153](#)
[graph\(\)](#) (in module [abjad.tools.topleveltools.graph](#)), [HTMLOutputFormat](#) (class in [ab-](#)
[1349](#) [jad.tools.abjadbooktools.HTMLOutputFormat](#),
[GraphvizEdge](#) (class in [ab-](#) [1383](#)
[jad.tools.documentationtools.GraphvizEdge](#)),
[1455](#)
[GraphvizField](#) (class in [ab-](#) [ImportManager](#) (class in [ab-](#)
[jad.tools.documentationtools.GraphvizField](#)), [jad.tools.systemtools.ImportManager](#),
[1457](#) [1703](#)
[GraphvizGraph](#) (class in [ab-](#) [ImpreciseTempoError](#) (class in [ab-](#)
[jad.tools.documentationtools.GraphvizGraph](#)), [jad.tools.exceptiontools.ImpreciseTempoError](#),
[1461](#) [1603](#)
[GraphvizGroup](#) (class in [ab-](#) [IncisedRhythmMaker](#) (class in [ab-](#)
[jad.tools.documentationtools.GraphvizGroup](#)), [jad.tools.rhythm makertools.IncisedRhythmMaker](#),
[1473](#) [824](#)
[GraphvizNode](#) (class in [ab-](#) [InciseSpecifier](#) (class in [ab-](#)
[jad.tools.documentationtools.GraphvizNode](#)), [jad.tools.rhythm makertools.InciseSpecifier](#),
[1482](#) [821](#)
[GraphvizObject](#) (class in [ab-](#) [increase_elements\(\)](#) (in module [ab-](#)
[jad.tools.documentationtools.GraphvizObject](#)), [jad.tools.sequencetools.increase_elements](#)),
[1433](#) [1089](#)
[GraphvizSubgraph](#) (class in [ab-](#) [IndicatorExpression](#) (class in [ab-](#)
[jad.tools.documentationtools.GraphvizSubgraph](#)), [jad.tools.indicator tools.IndicatorExpression](#)),
[1491](#) [179](#)
[greatest_common_divisor\(\)](#) (in module [ab-](#) [Infinity](#) (class in [abjad.tools.mathtools.Infinity](#)), 449
[jad.tools.mathtools.greatest_common_divisor](#)), [InheritanceGraph](#) (class in [ab-](#)
[472](#) [jad.tools.documentationtools.InheritanceGraph](#),
[greatest_multiple_less_equal\(\)](#) (in module [ab-](#) [1501](#)
[jad.tools.mathtools.greatest_multiple_less_equal](#)),
[472](#)
[greatest_power_of_two_less_equal\(\)](#) (in module [ab-](#) [insert_and_transpose_nested_subruns_in_pitch_class_number_list\(\)](#)
[jad.tools.mathtools.greatest_power_of_two_less_equal](#)), [\(in module \[ab-\]\(#\)](#)
[685](#) [jad.tools.pitchtools.insert_and_transpose_nested_subruns_in_p](#)

inspect_()	(in module abjad.tools.topleveltools.inspect_), 1350	is_fraction_equivalent_pair()	(in module abjad.tools.mathtools.is_fraction_equivalent_pair), 475
InspectionAgent	(class in abjad.tools.agenttools.InspectionAgent), 3	is_integer_equivalent_expr()	(in module abjad.tools.mathtools.is_integer_equivalent_expr), 476
instantiate_pitch_and_interval_test_collection()	(in module abjad.tools.pitchtools.instantiate_pitch_and_interval_test_collection), 685	is_integer_equivalent_n_tuple()	(in module abjad.tools.mathtools.is_integer_equivalent_n_tuple), 476
Instrument	(class in abjad.tools.instrumenttools.Instrument), 311	is_integer_equivalent_number()	(in module abjad.tools.mathtools.is_integer_equivalent_number), 476
InstrumentInventory	(class in abjad.tools.instrumenttools.InstrumentInventory), 313	is_integer_equivalent_pair()	(in module abjad.tools.mathtools.is_integer_equivalent_pair), 476
integer_equivalent_number_to_integer()	(in module abjad.tools.mathtools.integer_equivalent_number_to_integer), 474	is_integer_equivalent_singleton()	(in module abjad.tools.mathtools.is_integer_equivalent_singleton), 477
integer_to_base_k_tuple()	(in module abjad.tools.mathtools.integer_to_base_k_tuple), 474	is_integer_n_tuple()	(in module abjad.tools.mathtools.is_integer_n_tuple), 477
integer_to_binary_string()	(in module abjad.tools.mathtools.integer_to_binary_string), 474	is_integer_pair()	(in module abjad.tools.mathtools.is_integer_pair), 477
interlace_sequences()	(in module abjad.tools.sequencetools.interlace_sequences), 1089	is_integer_singleton()	(in module abjad.tools.mathtools.is_integer_singleton), 477
Interval	(class in abjad.tools.pitchtools.Interval), 503	is_lower_camel_case()	(in module abjad.tools.stringtools.is_lower_camel_case), 1191
IntervalClass	(class in abjad.tools.pitchtools.IntervalClass), 505	is_n_tuple()	(in module abjad.tools.mathtools.is_n_tuple), 478
IntervalClassSegment	(class in abjad.tools.pitchtools.IntervalClassSegment), 526	is_negative_integer()	(in module abjad.tools.mathtools.is_negative_integer), 478
IntervalClassSet	(class in abjad.tools.pitchtools.IntervalClassSet), 529	is_nonnegative_integer()	(in module abjad.tools.mathtools.is_nonnegative_integer), 478
IntervalClassVector	(class in abjad.tools.pitchtools.IntervalClassVector), 533	is_nonnegative_integer_equivalent_number()	(in module abjad.tools.mathtools.is_nonnegative_integer_equivalent_number), 478
IntervalSegment	(class in abjad.tools.pitchtools.IntervalSegment), 537	is_nonnegative_integer_power_of_two()	(in module abjad.tools.mathtools.is_nonnegative_integer_power_of_two), 479
IntervalSet	(class in abjad.tools.pitchtools.IntervalSet), 540	is_null_tuple()	(in module abjad.tools.mathtools.is_null_tuple), 479
IntervalVector	(class in abjad.tools.pitchtools.IntervalVector), 544	is_pair()	(in module abjad.tools.mathtools.is_pair), 479
inventory_aggregate_subsets()	(in module abjad.tools.pitchtools.inventory_aggregate_subsets), 686	is_positive_integer()	(in module abjad.tools.mathtools.is_positive_integer), 479
IOManager	(class in abjad.tools.systemtools.IOManager), 1699	is_positive_integer_equivalent_number()	(in module abjad.tools.mathtools.is_positive_integer_equivalent_number), 480
is_assignable_integer()	(in module abjad.tools.mathtools.is_assignable_integer), 474	is_positive_integer_power_of_two()	(in module abjad.tools.mathtools.is_positive_integer_power_of_two), 480
is_dash_case()	(in module abjad.tools.stringtools.is_dash_case), 1191		
is_dash_case_file_name()	(in module abjad.tools.stringtools.is_dash_case_file_name), 1191		
is_dotted_integer()	(in module abjad.tools.mathtools.is_dotted_integer), 474		

480
is_singleton() (in module abjad.tools.mathtools.is_singleton), 480
is_snake_case() (in module abjad.tools.stringtools.is_snake_case), 1192
is_snake_case_file_name() (in module abjad.tools.stringtools.is_snake_case_file_name), 1192
is_snake_case_file_name_with_extension() (in module abjad.tools.stringtools.is_snake_case_file_name_with_extension), 1192
is_snake_case_package_name() (in module abjad.tools.stringtools.is_snake_case_package_name), 1192
is_space_delimited_lowercase() (in module abjad.tools.stringtools.is_space_delimited_lowercase), 1193
is_string() (in module abjad.tools.stringtools.is_string), 1193
is_upper_camel_case() (in module abjad.tools.stringtools.is_upper_camel_case), 1193
IsAtSoundingPitch (class in abjad.tools.indicatortools.IsAtSoundingPitch), 181
IsUnpitched (class in abjad.tools.indicatortools.IsUnpitched), 182
iterate() (in module abjad.tools.topleveltools.iterate), 1350
iterate_named_pitch_pairs_in_expr() (in module abjad.tools.pitchtools.iterate_named_pitch_pairs_in_expr), 686
iterate_out_of_range_notes_and_chords() (in module abjad.tools.instrumenttools.iterate_out_of_range_notes_and_chords), 387
iterate_sequence_boustrophedon() (in module abjad.tools.sequencetools.iterate_sequence_boustrophedon), 1089
iterate_sequence_nwise() (in module abjad.tools.sequencetools.iterate_sequence_nwise), 1090
IterationAgent (class in abjad.tools.agenttools.IterationAgent), 9

J

```

JobHandler          (class          in          ab-
                    jad.tools.quantizationtools.JobHandler),
502
join_subsequences() (in          module          ab-
                    jad.tools.sequencetools.join_subsequences),
1091
join_subsequences_by_sign_of_elements()
(in          module          ab-
jad.tools.sequencetools.join_subsequences_by
1091

```

K

KeyCluster (class in abjad.tools.indicator.tools.KeyCluster), 183

KeySignature (class in abjad.tools.indicator.tools.KeySignature), 185

L

label_leaves_in_expr_with_leaf_depth() (in module abjad.tools.labeltools.label_leaves_in_expr_with_leaf_depth),
394
label_leaves_in_expr_with_leaf_duration() (in module abjad.tools.labeltools.label_leaves_in_expr_with_leaf_duration),
395
label_leaves_in_expr_with_leaf_durations() (in module abjad.tools.labeltools.label_leaves_in_expr_with_leaf_durations),
395
label_leaves_in_expr_with_leaf_indices() (in module abjad.tools.labeltools.label_leaves_in_expr_with_leaf_indices),
396
label_leaves_in_expr_with_leaf_numbers() (in module abjad.tools.labeltools.label_leaves_in_expr_with_leaf_numbers),
396
label_leaves_in_expr_with_named_interval_classes() (in module abjad.tools.labeltools.label_leaves_in_expr_with_named_interval_classes),
397
label_leaves_in_expr_with_named_intervals() (in module abjad.tools.labeltools.label_leaves_in_expr_with_named_intervals),
397
label_leaves_in_expr_with_numbered_interval_classes() (in module abjad.tools.labeltools.label_leaves_in_expr_with_numbered_interval_classes),
397
label_leaves_in_expr_with_numbered_intervals() (in module abjad.tools.labeltools.label_leaves_in_expr_with_numbered_intervals),
398
label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes() (in module abjad.tools.labeltools.label_leaves_in_expr_with_numbered_inversion_equivalent_interval_classes),
398
label_leaves_in_expr_with_pitch_class_numbers() (in module abjad.tools.labeltools.label_leaves_in_expr_with_pitch_class_numbers),
398
label_leaves_in_expr_with_pitch_numbers() (in module abjad.tools.labeltools.label_leaves_in_expr_with_pitch_numbers),
399
label_leaves_in_expr_with_tuplet_depth() (in module abjad.tools.labeltools.label_leaves_in_expr_with_tuplet_depth),
399

399	least_power_of_two_greater_equal()	(in module ab-	
label_leaves_in_expr_with_written_leaf_duration()	jad.tools.mathtools.least_power_of_two_greater_equal(),		
(in module ab-	481		
jad.tools.labeltools.label_leaves_in_expr_with_written_leaf_duration()	LilyPondCommand	(class in ab-	
400	jad.tools.indicatorstools.LilyPondCommand),		
label_logical_ties_in_expr_with_logical_tie_duration()	188		
(in module ab-	LilyPondComment	(class in ab-	
jad.tools.labeltools.label_logical_ties_in_expr_with_logical_tie_duration()	jad.tools.indicatorstools.LilyPondComment),		
400	190		
label_logical_ties_in_expr_with_logical_tie_durations()	LilyPondContextSetting	(class in ab-	
(in module ab-	jad.tools.lilypondnametools.LilyPondContextSetting),		
jad.tools.labeltools.label_logical_ties_in_expr_with_logical_tie_durations()	401		
401	LilyPondDimension	(class in ab-	
label_logical_ties_in_expr_with_written_logical_tie_duration()	jad.tools.lilypondfiletools.LilyPondDimension),		
(in module ab-	420		
jad.tools.labeltools.label_logical_ties_in_expr_with_written_logical_tie_duration()	LilyPondDuration	(class in ab-	
401	jad.tools.lilypondparsertools.LilyPondDuration),		
label_notes_in_expr_with_note_indices()	1632		
(in module ab-	LilyPondEvent	(class in ab-	
jad.tools.labeltools.label_notes_in_expr_with_note_indices()	jad.tools.lilypondparsertools.LilyPondEvent),		
401	1633		
label_vertical_moments_in_expr_with_interval_class_vectors()	LilyPondFile	(class in ab-	
(in module ab-	jad.tools.lilypondfiletools.LilyPondFile),		
jad.tools.labeltools.label_vertical_moments_in_expr_with_interval_class_vectors()	402		
402	LilyPondFormatBundle	(class in ab-	
label_vertical_moments_in_expr_with_named_intervals()	jad.tools.systemtools.LilyPondFormatBundle),		
(in module ab-	1704		
jad.tools.labeltools.label_vertical_moments_in_expr_with_named_intervals()	LilyPondFormatManager	(class in ab-	
402	jad.tools.systemtools.LilyPondFormatManager),		
label_vertical_moments_in_expr_with_numbered_interval_classes()	1706		
(in module ab-	LilyPondFraction	(class in ab-	
jad.tools.labeltools.label_vertical_moments_in_expr_with_numbered_interval_classes()	jad.tools.lilypondparsertools.LilyPondFraction),		
403	1634		
label_vertical_moments_in_expr_with_numbered_intervals()	LilyPondGrammarGenerator	(class in ab-	
(in module ab-	jad.tools.lilypondparsertools.LilyPondGrammarGenerator),		
jad.tools.labeltools.label_vertical_moments_in_expr_with_numbered_intervals()	403		
403	LilyPondGrobNameManager	(class in ab-	
label_vertical_moments_in_expr_with_numbered_pitch_classes()	jad.tools.lilypondnametools.LilyPondGrobNameManager),		
(in module ab-	1617		
jad.tools.labeltools.label_vertical_moments_in_expr_with_numbered_pitch_classes()	LilyPondGrobOverride	(class in ab-	
404	jad.tools.lilypondnametools.LilyPondGrobOverride),		
label_vertical_moments_in_expr_with_pitch_numbers()	1618		
(in module ab-	LilyPondLanguageToken	(class in ab-	
jad.tools.labeltools.label_vertical_moments_in_expr_with_pitch_numbers()	jad.tools.lilypondfiletools.LilyPondLanguageToken),		
404	425		
LaissezVibrer	(class in ab-	LilyPondLexicalDefinition	(class in ab-
jad.tools.indicatorstools.LaissezVibrer),	jad.tools.lilypondparsertools.LilyPondLexicalDefinition),		
187	1637		
LaTeXOutputFormat	(class in ab-	LilyPondNameManager	(class in ab-
jad.tools.abjadbooktools.LaTeXOutputFormat),	jad.tools.lilypondnametools.LilyPondNameManager),		
1385	1621		
Leaf (class in abjad.tools.scoretools.Leaf),	907	LilyPondParser	(class in ab-
least_common_multiple()	(in module ab-	jad.tools.lilypondparsertools.LilyPondParser),	
jad.tools.mathtools.least_common_multiple(),	481		
481	LilyPondParserError	(class in ab-	
least_multiple_greater_equal()	(in module ab-	jad.tools.exceptionstools.LilyPondParserError),	
jad.tools.mathtools.least_multiple_greater_equal(),	1604		
481	LilyPondSettingNameManager	(class in ab-	

jad.tools.lilypondnametools.LilyPondSettingNameManager),
 1622
 LilyPondSyntacticalDefinition (class in abjad.tools.lilypondfiletools.LilyPondSyntacticalDefinition),
 jad.tools.lilypondparsertools.LilyPondSyntacticalDefinition), 1650
 1650
 LilyPondVersionToken (class in abjad.tools.lilypondfiletools.LilyPondVersionToken),
 jad.tools.markuptools.make_big_centered_page_number_markup(), 426
 444
 Lineage (class in abjad.tools.selectiontools.Lineage), make_blank_line_markup() (in module abjad.tools.markuptools.make_blank_line_markup),
 1056
 list_all_abjad_classes() (in module abjad.tools.documentationtools.list_all_abjad_classes), 445
 1597
 list_all_abjad_functions() (in module abjad.tools.documentationtools.list_all_abjad_functions),
 jad.tools.markuptools.make_centered_title_markup(), 445
 1598
 list_all_classes() (in module abjad.tools.documentationtools.list_all_classes),
 jad.tools.spannertools.make_colored_text_spanner_with_nibs(), 1184
 1598
 list_all_experimental_classes() (in module abjad.tools.documentationtools.list_all_experimental_classes),
 jad.demos.desordre.make_desordre_cell(), 1355
 1598
 list_all_scoremanager_classes() (in module abjad.tools.documentationtools.list_all_scoremanager_classes),
 jad.demos.desordre.make_desordre_lilypond_file(), 1355
 1598
 list_all_scoremanager_functions() (in module abjad.tools.documentationtools.list_all_scoremanager_functions),
 jad.demos.desordre.make_desordre_measure(), 1355
 1598
 list_named_pitches_in_expr() (in module abjad.tools.pitchtools.list_named_pitches_in_expr),
 jad.demos.desordre.make_desordre_pitches(), 1355
 687
 list_numbered_interval_numbers_pairwise() (in module abjad.tools.pitchtools.list_numbered_interval_numbers_pairwise),
 jad.demos.desordre.make_desordre_score(), 1355
 687
 list_numbered_inversion_equivalent_interval_classes_pairwise() (in module abjad.tools.pitchtools.list_numbered_inversion_equivalent_interval_classes_pairwise),
 jad.demos.desordre.make_desordre_staff(), 1355
 688
 list_octave_transpositions_of_pitch_carrier_within_pitch_range() (in module abjad.tools.pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range),
 jad.tools.spannertools.make_dynamic_spanner_below_with_nib_at_right(), 1184
 689
 list_ordered_named_pitch_pairs_from_expr_1_to_expr_2() (in module abjad.tools.scoretools.make_empty_piano_score),
 jad.tools.scoretools.make_floating_time_signature_lilypond_file(), 1038
 689
 list_pitch_numbers_in_expr() (in module abjad.tools.pitchtools.list_pitch_numbers_in_expr),
 jad.tools.scoretools.make_leaves(), 1038
 690
 list_unordered_named_pitch_pairs_in_expr() (in module abjad.tools.scoretools.make_leaves_from_talea),
 jad.tools.scoretools.make_leaves_from_talea(), 1041
 690
 LogicalTie (class in abjad.tools.selectiontools.LogicalTie),
 jad.tools.documentationtools.make_ligeti_example_lilypond_file(), 1598
 1059
 make_lilypond_file() (in module abjad.demos.ferneyhough.make_lilypond_file),
 1357

make_lilypond_file()	(in module abjad.tools.rhythmmakertools.make_lilypond_file),	jad.tools.scoretools.make_repeated_notes_from_time_signature	1045
856		make_repeated_notes_from_time_signatures()	
make_mozart_lilypond_file()	(in module abjad.demos.mozart.make_mozart_lilypond_file),	jad.tools.scoretools.make_repeated_notes_from_time_signature	1045
1359			
make_mozart_measure()	(in module abjad.demos.mozart.make_mozart_measure),	make_repeated_notes_with_shorter_notes_at_end()	
1359		(in module abjad.tools.scoretools.make_repeated_notes_with_shorter_notes_at_end)	1045
make_mozart_measure_corpus()	(in module abjad.demos.mozart.make_mozart_measure_corpus)	make_repeated_rests_from_time_signatures()	
1359		(in module abjad.tools.scoretools.make_repeated_rests_from_time_signatures)	1046
make_mozart_score()	(in module abjad.demos.mozart.make_mozart_score),	make_repeated_skips_from_time_signatures()	
1359		(in module abjad.tools.scoretools.make_repeated_skips_from_time_signatures)	1046
make_multimeasure_rests()	(in module abjad.tools.scoretools.make_multimeasure_rests),	make_rests()	(in module abjad.tools.scoretools.make_rests), 1046
1042		make_rhythmic_sketch_staff()	(in module abjad.tools.scoretools.make_rhythmic_sketch_staff),
make_multiplied_quarter_notes()	(in module abjad.tools.scoretools.make_multiplied_quarter_notes),	1047	
1042		make_row_of_nested_tuplets()	(in module abjad.demos.ferneyhough.make_row_of_nested_tuplets),
make_n_middle_c_centered_pitches()	(in module abjad.tools.pitchtools.make_n_middle_c_centered_pitches),	1357	
690		make_rows_of_nested_tuplets()	(in module abjad.demos.ferneyhough.make_rows_of_nested_tuplets),
make_nested_tuplet()	(in module abjad.demos.ferneyhough.make_nested_tuplet),	1357	
1357		make_score()	(in module abjad.demos.ferneyhough.make_score), 1357
make_notes()	(in module abjad.tools.scoretools.make_notes), 1042	make_skips_with_multiplied_durations()	
make_notes_with_multiplied_durations()	(in module abjad.tools.scoretools.make_notes_with_multiplied_durations),	1043	
1043		make_part_lilypond_file()	(in module abjad.demos.part.make_part_lilypond_file),
make_part_lilypond_file()	(in module abjad.demos.part.make_part_lilypond_file),	1047	
1364		make_solid_text_spanner_with_nib()	(in module abjad.tools.spannertools.make_solid_text_spanner_with_nib),
make_percussion_note()	(in module abjad.tools.scoretools.make_percussion_note),	1185	
1043		make_spacer_skip_measures()	(in module abjad.tools.scoretools.make_spacer_skip_measures),
make_piano_score_from_leaves()	(in module abjad.tools.scoretools.make_piano_score_from_leaves),	1047	
1044		make_spacing_vector()	(in module abjad.tools.layouttools.make_spacing_vector),
make_piano_sketch_score_from_leaves()	(in module abjad.tools.scoretools.make_piano_sketch_score_from_leaves),	409	
1044		make_test_time_segments()	(in module abjad.tools.quantizationtools.make_test_time_segments),
make_reference_manual_graphviz_graph()	(in module abjad.tools.documentationtools.make_reference_manual_graphviz_graph),	800	
1598		make_text_alignment_example_lilypond_file()	(in module abjad.tools.documentationtools.make_text_alignment_example_lilypond_file),
make_reference_manual_lilypond_file()	(in module abjad.tools.documentationtools.make_reference_manual_lilypond_file),	1599	
1599		make_tied_leaf()	(in module abjad.tools.scoretools.make_tied_leaf), 1047
make_repeated_notes()	(in module abjad.tools.scoretools.make_repeated_notes),	make_vertically_adjusted_composer_markup()	
1044		(in module abjad.tools.markuptools.make_vertically_adjusted_composer_markup)	445
make_repeated_notes_from_time_signature()	(in module abjad.tools.scoretools.make_repeated_notes_from_time_signature)		
		MakeNewClassTemplateScript	(class in abjad.tools.scoretools)

jad.tools.developerscripttools.MakeNewClassTemplateScript(), 1412	module ab-
MakeNewFunctionTemplateScript (class in ab- jad.tools.developerscripttools.MakeNewFunctionTemplateScript), 1414	jad.tools.scoretools.move_full_measure_tuplet_prolation_to_m 1048
Marimba (class in ab- jad.tools.instrumenttools.Marimba), 318	move_full_measure_tuplet_prolation_to_full_measure_tuplet() (in module ab- jad.tools.scoretools.move_measure_prolation_to_full_measure 1049
Markup (class in abjad.tools.markuptools.Markup), 431	MultimeasureRest (class in ab- jad.tools.scoretools.MultimeasureRest), 972
MarkupCommand (class in ab- jad.tools.markuptools.MarkupCommand), 434	MultipartBeam (class in ab- jad.tools.spannertools.MultipartBeam), 1159
MarkupInventory (class in ab- jad.tools.markuptools.MarkupInventory), 437	Multiplier (class in ab- jad.tools.durationtools.Multiplier), 133
Matrix (class in abjad.tools.datastructuretools.Matrix), 65	Music (class in abjad.tools.lilypondparsertools.Music), 1625
Measure (class in abjad.tools.scoretools.Measure), 962	MusicGlyph (class in ab- jad.tools.markuptools.MusicGlyph), 442
MeasuredComplexBeam (class in ab- jad.tools.spannertools.MeasuredComplexBeam), 1155	mutate() (in module abjad.tools.topleveltools.mutate), 1350
MeasurewiseAttackPointOptimizer (class in ab- jad.tools.quantizationtools.MeasurewiseAttackPointOptimizer), 729	MutationAgent (class in ab- jad.tools.agenttools.MutationAgent), 18
MeasurewiseQSchema (class in ab- jad.tools.quantizationtools.MeasurewiseQSchema), 731	NaiveAttackPointOptimizer (class in ab- jad.tools.quantizationtools.NaiveAttackPointOptimizer), 739
MeasurewiseQSchemaItem (class in ab- jad.tools.quantizationtools.MeasurewiseQSchemaItem), 735	named_pitch_and_clef_to_staff_position_number() (in module ab- jad.tools.pitchtools.named_pitch_and_clef_to_staff_position_n 691
MeasurewiseQTarget (class in ab- jad.tools.quantizationtools.MeasurewiseQTarget), 737	NamedInterval (class in ab- jad.tools.pitchtools.NamedInterval), 548
Memoize (class in abjad.tools.systemtools.Memoize), 1708	NamedIntervalClass (class in ab- jad.tools.pitchtools.NamedIntervalClass), 553
Meter (class in abjad.tools.metertools.Meter), 489	NamedInversionEquivalentIntervalClass (class in ab- jad.tools.pitchtools.NamedInversionEquivalentIntervalClass), 555
MeterManager (class in ab- jad.tools.metertools.MeterManager), 497	NamedPitch (class in ab- jad.tools.pitchtools.NamedPitch), 558
MetricAccentKernel (class in ab- jad.tools.metertools.MetricAccentKernel), 499	NamedPitchClass (class in ab- jad.tools.pitchtools.NamedPitchClass), 565
MezzoSopranoVoice (class in ab- jad.tools.instrumenttools.MezzoSopranoVoice), 321	negate_elements() (in module ab- jad.tools.sequencetools.negate_elements), 1092
MissingMeasureError (class in ab- jad.tools.exceptiontools.MissingMeasureError), 1605	NegativeInfinity (class in ab- jad.tools.mathtools.NegativeInfinity), 451
MissingSpannerError (class in ab- jad.tools.exceptiontools.MissingSpannerError), 1606	new() (in module abjad.tools.topleveltools.new), 1351
MissingTempoError (class in ab- jad.tools.exceptiontools.MissingTempoError), 1607	next_integer_partition() (in module ab- jad.tools.mathtools.next_integer_partition), 482
Mode (class in abjad.tools.tonalanalysisistools.Mode), 1321	NonreducedFraction (class in ab- jad.tools.mathtools.NonreducedFraction), 453
ModuleCrawler (class in ab- jad.tools.documentationtools.ModuleCrawler), 1504	NonreducedRatio (class in ab-
move_full_measure_tuplet_prolation_to_measure_time_signature()	

- jad.tools.mathtools.NonreducedRatio),
460
- Note (class in abjad.tools.scoretools.Note), 974
- NoteHead (class in abjad.tools.scoretools.NoteHead),
977
- NoteHeadInventory (class in abjad.tools.scoretools.NoteHeadInventory),
980
- NoteRhythmMaker (class in abjad.tools.rhythm makertools.NoteRhythmMaker),
828
- notes_and_chords_are_in_range() (in module abjad.tools.instrumenttools.notes_and_chords_are_in_range),
388
- notes_and_chords_are_on_expected_clefs() (in module abjad.tools.instrumenttools.notes_and_chords_are_on_expected_clefs),
388
- NullAttackPointOptimizer (class in abjad.tools.quantizationtools.NullAttackPointOptimizer),
740
- NullContextManager (class in abjad.tools.systemtools.NullContextManager),
1710
- numbered_inversion_equivalent_interval_class_dictionary() (in module abjad.tools.pitchtools.numbered_inversion_equivalent_interval_class_dictionary),
691
- NumberedInterval (class in abjad.tools.pitchtools.NumberedInterval),
570
- NumberedIntervalClass (class in abjad.tools.pitchtools.NumberedIntervalClass),
573
- NumberedInversionEquivalentIntervalClass (class in abjad.tools.pitchtools.NumberedInversionEquivalentIntervalClass),
575
- NumberedPitch (class in abjad.tools.pitchtools.NumberedPitch), 578
- NumberedPitchClass (class in abjad.tools.pitchtools.NumberedPitchClass),
584
- NumberedPitchClassColorMap (class in abjad.tools.pitchtools.NumberedPitchClassColorMap),
589
- O**
- Oboe (class in abjad.tools.instrumenttools.Oboe), 324
- OctavationSpanner (class in abjad.tools.spannertools.OctavationSpanner),
1161
- Octave (class in abjad.tools.pitchtools.Octave), 591
- OctaveTranspositionMapping (class in abjad.tools.pitchtools.OctaveTranspositionMapping),
595
- OctaveTranspositionMappingComponent (class in abjad.tools.pitchtools.OctaveTranspositionMappingComponent),
600
- OctaveTranspositionMappingInventory (class in abjad.tools.pitchtools.OctaveTranspositionMappingInventory),
602
- Offset (class in abjad.tools.durationtools.Offset), 145
- offset_happens_after_timespan_starts() (in module abjad.tools.timespantools.offset_happens_after_timespan_starts),
1299
- offset_happens_after_timespan_stops() (in module abjad.tools.timespantools.offset_happens_after_timespan_stops),
1299
- offset_happens_before_timespan_starts() (in module abjad.tools.timespantools.offset_happens_before_timespan_starts),
1300
- offset_happens_before_timespan_stops() (in module abjad.tools.timespantools.offset_happens_before_timespan_stops),
1301
- offset_happens_during_timespan() (in module abjad.tools.timespantools.offset_happens_during_timespan),
1301
- offset_happens_when_timespan_starts() (in module abjad.tools.timespantools.offset_happens_when_timespan_starts),
1302
- offset_happens_when_timespan_stops() (in module abjad.tools.timespantools.offset_happens_when_timespan_stops),
1302
- OffsetTimespanTimeRelation (class in abjad.tools.timespantools.OffsetTimespanTimeRelation),
1237
- OrdinalConstant (class in abjad.tools.datastructuretools.OrdinalConstant),
68
- OutputFormat (class in abjad.tools.abjadbooktools.OutputFormat),
1608
- OverfullContainerError (class in abjad.tools.exceptiontools.OverfullContainerError),
1608
- override() (in module abjad.tools.topleveltools.override), 1351
- overwrite_elements() (in module abjad.tools.sequencetools.overwrite_elements),
1093
- P**
- ParallelJobHandler (class in abjad.tools.quantizationtools.ParallelJobHandler),
742
- ParallelJobHandlerWorker (class in abjad.tools.quantizationtools.ParallelJobHandlerWorker),
743
- Parentage (class in abjad.tools.selectiontools.Parentage), 1063
- parse() (in module abjad.tools.topleveltools.parse),
1351
- parse_ly_syntax() (in module abjad.tools.ly_syntax),
1351

jad.tools.lilypondparsertools.parse_reduced_ly_syntax), jad.tools.instrumenttools.Performer), 327
 1680
 PerformerInventory (class in ab-
 parse_rtm_syntax() (in module ab- jad.tools.instrumenttools.PerformerInventory),
 jad.tools.rhythmtreetools.parse_rtm_syntax), 334
 886
 permute_named_pitch_carrier_list_by_twelve_tone_row()
 Parser (class in abjad.tools.abctools.Parser), 1369
 (in module ab-
 PartCantusScoreTemplate (class in ab- jad.tools.pitchtools.permute_named_pitch_carrier_list_by_twel
 jad.demos.part.PartCantusScoreTemplate), 691
 1361
 permute_sequence() (in module ab-
 partition_integer_by_ratio() (in module ab- jad.tools.sequencetools.permute_sequence),
 jad.tools.mathtools.partition_integer_by_ratio), 1098
 482
 persist() (in module abjad.tools.topleveltools.persist),
 partition_integer_into_canonic_parts() (in module ab- 1351
 jad.tools.mathtools.partition_integer_into_canonic_parts), 483
 PersistenceAgent (class in ab-
 partition_integer_into_halves() (in module ab- jad.tools.agenttools.PersistenceAgent),
 jad.tools.mathtools.partition_integer_into_halves), 34
 484
 PhrasingSlur (class in ab-
 partition_integer_into_parts_less_than_double() Piano (class in abjad.tools.instrumenttools.Piano), 339
 (in module ab- PianoPedalSpanner (class in ab-
 jad.tools.mathtools.partition_integer_into_parts_less_than_double), 1167
 484
 piccolo (class in abjad.tools.instrumenttools.Piccolo),
 partition_integer_into_units() (in module ab- 342
 jad.tools.mathtools.partition_integer_into_units), 485
 Pipe (class in abjad.tools.documentationtools.Pipe),
 partition_sequence_by_counts() (in module ab- 1506
 jad.tools.sequencetools.partition_sequence_by_counts), 1093
 Pitch (class in abjad.tools.pitchtools.Pitch), 507
 PitchArray (class in abjad.tools.pitchtools.PitchArray),
 partition_sequence_by_ratio_of_lengths() 607
 (in module ab- PitchArrayCell (class in ab-
 jad.tools.sequencetools.partition_sequence_by_ratio_of_lengths), 613
 1094
 PitchArrayColumn (class in ab-
 partition_sequence_by_ratio_of_weights() jad.tools.pitchtools.PitchArrayColumn),
 (in module ab- 616
 jad.tools.sequencetools.partition_sequence_by_ratio_of_weights), 1094
 PitchArrayInventory (class in ab-
 partition_sequence_by_restricted_growth_function() jad.tools.pitchtools.PitchArrayInventory),
 (in module ab- 619
 jad.tools.sequencetools.partition_sequence_by_restricted_growth_function), 1095
 PitchArrayRow (class in ab-
 partition_sequence_by_restricted_growth_function() jad.tools.pitchtools.PitchArrayRow), 624
 1095
 PitchClass (class in abjad.tools.pitchtools.PitchClass),
 partition_sequence_by_sign_of_elements() 511
 (in module ab- PitchClassSegment (class in ab-
 jad.tools.sequencetools.partition_sequence_by_sign_of_elements), 629
 1095
 PitchClassSet (class in ab-
 partition_sequence_by_value_of_elements() jad.tools.pitchtools.PitchClassSet), 634
 (in module ab- PitchClassTree (class in ab-
 jad.tools.sequencetools.partition_sequence_by_value_of_elements), 639
 1096
 PitchClassVector (class in ab-
 partition_sequence_by_weights() (in module ab- jad.tools.pitchtools.PitchClassVector),
 jad.tools.sequencetools.partition_sequence_by_weights), 653
 1096
 PitchedQEvent (class in ab-
 PartitionError (class in ab- jad.tools.quantizationtools.PitchedQEvent),
 jad.tools.exceptiontools.PartitionError), 745
 1609
 PitchRange (class in ab-
 PayloadTree (class in ab- jad.tools.pitchtools.PitchRange), 656
 jad.tools.datastructuretools.PayloadTree), 70
 PitchRangeInventory (class in ab-
 Performer (class in ab- jad.tools.pitchtools.PitchRangeInventory),

660
PitchSegment (class in abjad.tools.pitchtools.PitchSegment), 665
PitchSet (class in abjad.tools.pitchtools.PitchSet), 671
PitchVector (class in abjad.tools.pitchtools.PitchVector), 675
play() (in module abjad.tools.topleveltools.play), 1351
pluralize() (in module abjad.tools.stringtools.pluralize), 1193
ProgressIndicator (class in abjad.tools.systemtools.ProgressIndicator), 1711
PyTestScript (class in abjad.tools.developerscripttools.PyTestScript), 1417

Q

QEvent (class in abjad.tools.quantizationtools.QEvent), 704
QEventProxy (class in abjad.tools.quantizationtools.QEventProxy), 747
QEventSequence (class in abjad.tools.quantizationtools.QEventSequence), 749
QGrid (class in abjad.tools.quantizationtools.QGrid), 755
QGridContainer (class in abjad.tools.quantizationtools.QGridContainer), 758
QGridLeaf (class in abjad.tools.quantizationtools.QGridLeaf), 771
QSchema (class in abjad.tools.quantizationtools.QSchema), 706
QSchemaItem (class in abjad.tools.quantizationtools.QSchemaItem), 708
QTarget (class in abjad.tools.quantizationtools.QTarget), 710
QTargetBeat (class in abjad.tools.quantizationtools.QTargetBeat), 777
QTargetMeasure (class in abjad.tools.quantizationtools.QTargetMeasure), 780
QuantizationJob (class in abjad.tools.quantizationtools.QuantizationJob), 783
Quantizer (class in abjad.tools.quantizationtools.Quantizer), 786

R

Ratio (class in abjad.tools.mathtools.Ratio), 463
RedirectedStreams (class in abjad.tools.systemtools.RedirectedStreams),

1713
ReducedLyParser (class in abjad.tools.lilypondparsertools.ReducedLyParser), 1668
register_pitch_class_numbers_by_pitch_number_aggregate() (in module abjad.tools.pitchtools.register_pitch_class_numbers_by_pitch_number_aggregate), 691
remove_elements() (in module abjad.tools.sequencetools.remove_elements), 1098
remove_markup_from_leaves_in_expr() (in module abjad.tools.labeltools.remove_markup_from_leaves_in_expr), 405
remove_powers_of_two() (in module abjad.tools.mathtools.remove_powers_of_two), 485
remove_repeated_elements() (in module abjad.tools.sequencetools.remove_repeated_elements), 1099
remove_subsequence_of_weight_at_index() (in module abjad.tools.sequencetools.remove_subsequence_of_weight_at_index), 1100
RenameModulesScript (class in abjad.tools.developerscripttools.RenameModulesScript), 1420
repeat_elements() (in module abjad.tools.sequencetools.repeat_elements), 1100
repeat_sequence() (in module abjad.tools.sequencetools.repeat_sequence), 1101
repeat_sequence_to_length() (in module abjad.tools.sequencetools.repeat_sequence_to_length), 1101
repeat_sequence_to_weight() (in module abjad.tools.sequencetools.repeat_sequence_to_weight), 1101
replace_elements() (in module abjad.tools.sequencetools.replace_elements), 1102
ReplaceInFilesScript (class in abjad.tools.developerscripttools.ReplaceInFilesScript), 1423
requires() (in module abjad.tools.systemtools.requires), 1727
ResidueClass (class in abjad.tools.sievetools.ResidueClass), 1115
Rest (class in abjad.tools.scoretools.Rest), 985
ReSTAutodocDirective (class in abjad.tools.documentationtools.ReSTAutodocDirective), 1508
ReSTAutosummaryDirective (class in abjad.tools.documentationtools.ReSTAutosummaryDirective), 1518
ReSTAutosummaryItem (class in abjad.tools.documentationtools.ReSTAutosummaryItem),

1527	RomanNumeral (class in abjad.tools.tonalanalysistools.RomanNumeral), 1323
ReSTDirective (class in abjad.tools.documentationtools.ReSTDirective), 1435	RootedChordClass (class in abjad.tools.tonalanalysistools.RootedChordClass), 1325
ReSTDDocument (class in abjad.tools.documentationtools.ReSTDDocument), 1531	RootlessChordClass (class in abjad.tools.tonalanalysistools.RootlessChordClass), 1331
ReSTHeading (class in abjad.tools.documentationtools.ReSTHeading), 1541	rotate_sequence() (in module abjad.tools.sequencetools.rotate_sequence), 1104
ReSTHorizontalRule (class in abjad.tools.documentationtools.ReSTHorizontalRule), 1545	run_abjad() (in module abjad.tools.systemtools.run_abjad), 1728
ReSTInheritanceDiagram (class in abjad.tools.documentationtools.ReSTInheritanceDiagram), 1549	run_abjadbook() (in module abjad.tools.developerscripttools.run_abjadbook), 1431
ReSTLineageDirective (class in abjad.tools.documentationtools.ReSTLineageDirective), 1558	run_ajv() (in module abjad.tools.developerscripttools.run_ajv), 1431
ReSTOnlyDirective (class in abjad.tools.documentationtools.ReSTOnlyDirective), 1568	RunDoctestScript (class in abjad.tools.developerscripttools.RunDoctestScript), 1426
ReSTOutputFormat (class in abjad.tools.abjadbooktools.ReSTOutputFormat), 1387	S
ReSTParagraph (class in abjad.tools.documentationtools.ReSTParagraph), 1578	Scale (class in abjad.tools.tonalanalysistools.Scale), 1335
RestRhythmMaker (class in abjad.tools.rhythmmakertools.RestRhythmMaker), 831	scale_measure_denominator_and_adjust_measure_contents() (in module abjad.tools.scoretools.scale_measure_denominator_and_adjust_m), 1049
ReSTTOCDirective (class in abjad.tools.documentationtools.ReSTTOCDirective), 1582	ScaleDegree (class in abjad.tools.tonalanalysistools.ScaleDegree), 1341
ReSTTOCItem (class in abjad.tools.documentationtools.ReSTTOCItem), 1591	Scheme (class in abjad.tools.schemetools.Scheme), 887
retain_elements() (in module abjad.tools.sequencetools.retain_elements), 1103	SchemeAssociativeList (class in abjad.tools.schemetools.SchemeAssociativeList), 890
reverse_sequence() (in module abjad.tools.sequencetools.reverse_sequence), 1104	SchemeColor (class in abjad.tools.schemetools.SchemeColor), 892
RhythmMaker (class in abjad.tools.rhythmmakertools.RhythmMaker), 801	SchemeMoment (class in abjad.tools.schemetools.SchemeMoment), 894
RhythmTreeContainer (class in abjad.tools.rhythmtreetools.RhythmTreeContainer), 863	SchemePair (class in abjad.tools.schemetools.SchemePair), 897
RhythmTreeLeaf (class in abjad.tools.rhythmtreetools.RhythmTreeLeaf), 877	SchemeParser (class in abjad.tools.lilypondparsertools.SchemeParser), 1674
RhythmTreeNode (class in abjad.tools.rhythmtreetools.RhythmTreeNode), 857	SchemeParserFinishedError (class in abjad.tools.exceptiontools.SchemeParserFinishedError), 1610
RhythmTreeParser (class in abjad.tools.rhythmtreetools.RhythmTreeParser), 883	SchemeVector (class in abjad.tools.schemetools.SchemeVector), 899
	SchemeVectorConstant (class in abjad.tools.schemetools.SchemeVectorConstant), 901
	Score (class in abjad.tools.scoretools.Score), 987

SearchTree	(class in abjad.tools.quantizationtools.SearchTree), 712	jad.tools.selectiontools.SimultaneousSelection), 1074
Segment	(class in abjad.tools.pitchtools.Segment), 514	Skip (class in abjad.tools.scoretools.Skip), 995
select()	(in module abjad.tools.tonalanalysistools.select), 1347	SkipRhythmMaker (class in abjad.tools.rhythm makertools.SkipRhythmMaker), 834
select()	(in module abjad.tools.topleveltools.select), 1352	SliceSelection (class in abjad.tools.selectiontools.SliceSelection), 1076
Selection	(class in abjad.tools.selectiontools.Selection), 1067	Slur (class in abjad.tools.spannertools.Slur), 1170
SelectionInventory	(class in abjad.tools.selectiontools.SelectionInventory), 1069	snake_case_to_lower_camel_case() (in module abjad.tools.stringtools.snake_case_to_lower_camel_case), 1194
Sequence	(class in abjad.tools.sequencetools.Sequence), 1083	snake_case_to_upper_camel_case() (in module abjad.tools.stringtools.snake_case_to_upper_camel_case), 1194
SequentialMusic	(class in abjad.tools.lilypondparsertools.SequentialMusic), 1678	SopranoSaxophone (class in abjad.tools.instrumenttools.SopranoSaxophone), 345
SerialJobHandler	(class in abjad.tools.quantizationtools.SerialJobHandler), 790	SopranoSaxophone (class in abjad.tools.instrumenttools.SopranoSaxophone), 348
Set	(class in abjad.tools.pitchtools.Set), 517	SopranoVoice (class in abjad.tools.instrumenttools.SopranoVoice), 351
set_()	(in module abjad.tools.topleveltools.set_), 1352	sort_named_pitch_carriers_in_expr() (in module abjad.tools.pitchtools.sort_named_pitch_carriers_in_expr), 692
set_line_breaks_by_line_duration()	(in module abjad.tools.layouttools.set_line_breaks_by_line_duration), 409	SortedCollection (class in abjad.tools.datastructuretools.SortedCollection), 84
set_line_breaks_by_line_duration_ge()	(in module abjad.tools.layouttools.set_line_breaks_by_line_duration_ge), 409	space_delimited_lower_case_to_upper_camel_case() (in module abjad.tools.stringtools.space_delimited_lowercase_to_upper_camel_case), 1194
set_line_breaks_by_line_duration_in_seconds_ge()	(in module abjad.tools.layouttools.set_line_breaks_by_line_duration_in_seconds_ge), 410	SpacingIndication (class in abjad.tools.layouttools.SpacingIndication), 407
set_measure_denominator_and_adjust_numerator()	(in module abjad.tools.scoretools.set_measure_denominator_and_adjust_numerator), 1049	Spanner (class in abjad.tools.spannertools.Spanner), 113
set_written_pitch_of_pitched_components_in_expr()	(in module abjad.tools.pitchtools.set_written_pitch_of_pitched_components_in_expr), 692	spell_numbered_interval_number() (in module abjad.tools.pitchtools.spell_numbered_interval_number), 692
shadow_pitch_contour_reservoir()	(in module abjad.demos.part.shadow_pitch_contour_reservoir), 1364	spell_pitch_number() (in module abjad.tools.pitchtools.spell_pitch_number), 692
show()	(in module abjad.tools.topleveltools.show), 1352	splice_between_elements() (in module abjad.tools.sequencetools.splice_between_elements), 1104
Sieve	(class in abjad.tools.sievetools.Sieve), 1118	split_sequence() (in module abjad.tools.sequencetools.split_sequence), 1105
sign()	(in module abjad.tools.mathtools.sign), 485	Staff (class in abjad.tools.scoretools.Staff), 997
SilentQEvent	(class in abjad.tools.quantizationtools.SilentQEvent), 791	StaffChange (class in abjad.tools.indicator tools.StaffChange), 192
SimpleInequality	(class in abjad.tools.timespantools.SimpleInequality), 1240	StaffGroup (class in abjad.tools.scoretools.StaffGroup), 1004
SimultaneousMusic	(class in abjad.tools.lilypondparsertools.SimultaneousMusic), 1627	StaffLinesSpanner (class in ab-
SimultaneousSelection	(class in ab-	

jad.tools.spannertools.StaffLinesSpanner), 1175	jad.tools.instrumenttools.TenorSaxophone), 354
StatalServer (class in ab- jad.tools.datastructuretools.StatalServer), 86	TenorTrombone (class in ab- jad.tools.instrumenttools.TenorTrombone), 357
StatalServerCursor (class in ab- jad.tools.datastructuretools.StatalServerCursor), 88	TenorVoice (class in ab- jad.tools.instrumenttools.TenorVoice), 360
StemTremolo (class in ab- jad.tools.indicatortools.StemTremolo), 194	TerminalQEvent (class in ab- jad.tools.quantizationtools.TerminalQEvent), 793
StorageFormatManager (class in ab- jad.tools.systemtools.StorageFormatManager), 1715	TestAndRebuildScript (class in ab- jad.tools.developerscripttools.TestAndRebuildScript), 1429
StorageFormatSpecification (class in ab- jad.tools.systemtools.StorageFormatSpecification), 1717	TestManager (class in ab- jad.tools.systemtools.TestManager), 1720
StringContactPoint (class in ab- jad.tools.indicatortools.StringContactPoint), 196	TextSpanner (class in ab- jad.tools.spannertools.TextSpanner), 1177
StringNumber (class in ab- jad.tools.indicatortools.StringNumber), 197	Tie (class in abjad.tools.spannertools.Tie), 1179
StringOrchestraScoreTemplate (class in ab- jad.tools.templatetools.StringOrchestraScoreTemplate), 1203	TieSpecifier (class in ab- jad.tools.rhythmmakertools.TieSpecifier), 849
StringQuartetScoreTemplate (class in ab- jad.tools.templatetools.StringQuartetScoreTemplate), 1205	Timer (class in abjad.tools.systemtools.Timer), 1722
strip_diacritics() (in module ab- jad.tools.stringtools.strip_diacritics), 1194	TimeRelation (class in ab- jad.tools.timespantools.TimeRelation), 1209
suggest_clef_for_named_pitches() (in module ab- jad.tools.pitchtools.suggest_clef_for_named_pitches), 693	TimeSignature (class in ab- jad.tools.indicatortools.TimeSignature), 209
sum_consecutive_elements_by_sign() (in module ab- jad.tools.sequencetools.sum_consecutive_elements_by_sign), 1105	TimeSignatureInventory (class in ab- jad.tools.indicatortools.TimeSignatureInventory), 213
sum_elements() (in module ab- jad.tools.sequencetools.sum_elements), 1106	Timespan (class in ab- jad.tools.timespantools.Timespan), 1242
SyntaxNode (class in ab- jad.tools.lilypondparsertools.SyntaxNode), 1679	timespan_2_contains_timespan_1_improperly() (in module ab- jad.tools.timespantools.timespan_2_contains_timespan_1_improperly), 1302
T	timespan_2_curtails_timespan_1() (in module ab- jad.tools.timespantools.timespan_2_curtails_timespan_1), 1303
Talea (class in abjad.tools.rhythmmakertools.Talea), 837	timespan_2_delays_timespan_1() (in module ab- jad.tools.timespantools.timespan_2_delays_timespan_1), 1303
TaleaRhythmMaker (class in ab- jad.tools.rhythmmakertools.TaleaRhythmMaker), 839	timespan_2_happens_during_timespan_1() (in module ab- jad.tools.timespantools.timespan_2_happens_during_timespan_1), 1304
Tempo (class in abjad.tools.indicatortools.Tempo), 199	timespan_2_intersects_timespan_1() (in module ab- jad.tools.timespantools.timespan_2_intersects_timespan_1), 1304
TempoInventory (class in ab- jad.tools.indicatortools.TempoInventory), 204	timespan_2_is_congruent_to_timespan_1() (in module ab- jad.tools.timespantools.timespan_2_is_congruent_to_timespan_1), 1304
TemporaryDirectoryChange (class in ab- jad.tools.systemtools.TemporaryDirectoryChange), 1719	timespan_2_overlaps_all_of_timespan_1() (in module ab- jad.tools.timespantools.timespan_2_overlaps_all_of_timespan_1), 1304
TenorSaxophone (class in ab-	

1305	jad.tools.timespantools.timespan_2_stops_before_timespan_1_
timespan_2_overlaps_only_start_of_timespan_1()	1310
(in module ab-	timespan_2_stops_during_timespan_1()) (in module ab-
jad.tools.timespantools.timespan_2_overlaps_only_start_of_timespan_1())	jad.tools.timespantools.timespan_2_stops_during_timespan_1()),
1305	1311
timespan_2_overlaps_only_stop_of_timespan_1()	timespan_2_stops_when_timespan_1_starts()
(in module ab-	(in module ab-
jad.tools.timespantools.timespan_2_overlaps_only_stop_of_timespan_1())	jad.tools.timespantools.timespan_2_stops_when_timespan_1_s
1305	1311
timespan_2_overlaps_start_of_timespan_1()	timespan_2_stops_when_timespan_1_stops()
(in module ab-	(in module ab-
jad.tools.timespantools.timespan_2_overlaps_start_of_timespan_1())	jad.tools.timespantools.timespan_2_stops_when_timespan_1_s
1306	1311
timespan_2_overlaps_stop_of_timespan_1()	timespan_2_trisects_timespan_1() (in module ab-
(in module ab-	jad.tools.timespantools.timespan_2_trisects_timespan_1),
jad.tools.timespantools.timespan_2_overlaps_stop_of_timespan_1()),	
1306	TimespanInventory (class in ab-
timespan_2_starts_after_timespan_1_starts()	jad.tools.timespantools.TimespanInventory),
(in module ab-	1262
jad.tools.timespantools.timespan_2_starts_after_timespan_1_starts())	TimespanTimeRelation (class in ab-
1306	jad.tools.timespantools.TimespanTimeRelation),
timespan_2_starts_after_timespan_1_stops()	1294
(in module ab-	to_accent_free_snake_case() (in module ab-
jad.tools.timespantools.timespan_2_starts_after_timespan_1_stops())	jad.tools.stringtools.to_accent_free_snake_case),
1307	1194
timespan_2_starts_before_timespan_1_starts()	to_dash_case() (in module ab-
(in module ab-	jad.tools.stringtools.to_dash_case), 1195
jad.tools.timespantools.timespan_2_starts_before_timespan_1_starts())	to_snake_case() (in module ab-
1307	jad.tools.stringtools.to_snake_case), 1195
timespan_2_starts_before_timespan_1_stops()	to_space_delimited_lowercase() (in module ab-
(in module ab-	jad.tools.stringtools.to_space_delimited_lowercase),
jad.tools.timespantools.timespan_2_starts_before_timespan_1_stops()),	
1307	96
timespan_2_starts_during_timespan_1() (in module ab-	to_upper_camel_case() (in module ab-
jad.tools.timespantools.timespan_2_starts_during_timespan_1())	jad.tools.stringtools.to_upper_camel_case),
1308	an 96,
timespan_2_starts_when_timespan_1_starts()	TonalAnalysisAgent (class in ab-
(in module ab-	jad.tools.tonalanalysistools.TonalAnalysisAgent),
jad.tools.timespantools.timespan_2_starts_when_timespan_1_starts())	1344
1309	ToolsPackageDocumenter (class in ab-
timespan_2_starts_when_timespan_1_stops()	jad.tools.documentationtools.ToolsPackageDocumenter),
(in module ab-	1595
jad.tools.timespantools.timespan_2_starts_when_timespan_1_stops())	transpose_from_sounding_pitch_to_written_pitch()
1309	(in module ab-
timespan_2_stops_after_timespan_1_starts()	389
(in module ab-	transpose_from_written_pitch_to_sounding_pitch()
jad.tools.timespantools.timespan_2_stops_after_timespan_1_starts())	(in module ab-
1309	jad.tools.instrumenttools.transpose_from_written_pitch_to_sou
timespan_2_stops_after_timespan_1_stops()	389
(in module ab-	transpose_named_pitch_by_numbered_interval_and_respell()
jad.tools.timespantools.timespan_2_stops_after_timespan_1_stops())	(in module ab-
1310	jad.tools.pitchtools.transpose_named_pitch_by_numbered_inte
timespan_2_stops_before_timespan_1_starts()	693
(in module ab-	transpose_pitch_carrier_by_interval() (in module ab-
jad.tools.timespantools.timespan_2_stops_before_timespan_1_starts())	jad.tools.pitchtools.transpose_pitch_carrier_by_interval),
1310	693
timespan_2_stops_before_timespan_1_stops()	transpose_pitch_class_number_to_pitch_number_neighbor()
(in module ab-	(in module ab-

[jad.tools.pitchtools.transpose_pitch_class_number_to_pitch_number_neighbor](#),
[693](#)
[transpose_pitch_expr_into_pitch_range\(\)](#) ([in module abjad.tools.pitchtools.transpose_pitch_expr_into_pitch_range](#)),
[694](#)
[transpose_pitch_number_by_octave_transposition_mapping\(\)](#) ([in module abjad.tools.pitchtools.transpose_pitch_number_by_octave_transposition_mapping](#)),
[694](#)
[TreeContainer](#) ([class in abjad.tools.datastructuretools.TreeContainer](#)),
[91](#)
[TreeNode](#) ([class in abjad.tools.datastructuretools.TreeNode](#)),
[100](#)
[TrillSpanner](#) ([class in abjad.tools.spannertools.TrillSpanner](#)), [1182](#)
[Trumpet](#) ([class in abjad.tools.instrumenttools.Trumpet](#)),
[363](#)
[truncate_sequence\(\)](#) ([in module abjad.tools.sequencetools.truncate_sequence](#)),
[1106](#)
[Tuba](#) ([class in abjad.tools.instrumenttools.Tuba](#)), [366](#)
[Tuning](#) ([class in abjad.tools.indicatortools.Tuning](#)), [218](#)
[Tuplet](#) ([class in abjad.tools.scoretools.Tuplet](#)), [1011](#)
[TupletRhythmMaker](#) ([class in abjad.tools.rhythmmakertools.TupletRhythmMaker](#)),
[851](#)
[TupletSpellingSpecifier](#) ([class in abjad.tools.rhythmmakertools.TupletSpellingSpecifier](#)),
[855](#)
[TwelveToneRow](#) ([class in abjad.tools.pitchtools.TwelveToneRow](#)),
[678](#)
[TwoStaffPianoScoreTemplate](#) ([class in abjad.tools.templateTools.TwoStaffPianoScoreTemplate](#)),
[1206](#)
[TypedCollection](#) ([class in abjad.tools.datastructuretools.TypedCollection](#)),
[39](#)
[TypedCounter](#) ([class in abjad.tools.datastructuretools.TypedCounter](#)),
[104](#)
[TypedFrozenSet](#) ([class in abjad.tools.datastructuretools.TypedFrozenSet](#)),
[107](#)
[TypedList](#) ([class in abjad.tools.datastructuretools.TypedList](#)),
[110](#)
[TypedOrderedDict](#) ([class in abjad.tools.datastructuretools.TypedOrderedDict](#)),
[115](#)
[TypedTuple](#) ([class in abjad.tools.datastructuretools.TypedTuple](#)),
[118](#)

[UnboundedTimeIntervalError](#) ([class in abjad.tools.exceptionTools.UnboundedTimeIntervalError](#)),
[1611](#)
[UnderfullContainerError](#) ([class in abjad.tools.exceptionTools.UnderfullContainerError](#)),
[1612](#)
[UntunedPercussion](#) ([class in abjad.tools.instrumentTools.UntunedPercussion](#)),
[369](#)
[UnweightedSearchTree](#) ([class in abjad.tools.quantizationTools.UnweightedSearchTree](#)),
[795](#)
[UpdateManager](#) ([class in abjad.tools.systemTools.UpdateManager](#)),
[1724](#)
[upper_camel_case_to_snake_case\(\)](#) ([in module abjad.tools.stringTools.upper_camel_case_to_snake_case](#)),
[1196](#)
[upper_camel_case_to_space_delimited_lowercase\(\)](#) ([in module abjad.tools.stringTools.upper_camel_case_to_space_delimited_lowercase](#)),
[1197](#)

V
[Vector](#) ([class in abjad.tools.pitchTools.Vector](#)), [520](#)
[VerticalMoment](#) ([class in abjad.tools.selectionTools.VerticalMoment](#)),
[1079](#)
[Vibraphone](#) ([class in abjad.tools.instrumentTools.Vibraphone](#)),
[372](#)
[Viola](#) ([class in abjad.tools.instrumentTools.Viola](#)), [375](#)
[Violin](#) ([class in abjad.tools.instrumentTools.Violin](#)), [378](#)
[Voice](#) ([class in abjad.tools.scoreTools.Voice](#)), [1028](#)

W
[weight\(\)](#) ([in module abjad.tools.mathTools.weight](#)), [486](#)
[WeightedSearchTree](#) ([class in abjad.tools.quantizationTools.WeightedSearchTree](#)),
[798](#)
[WellformednessManager](#) ([class in abjad.tools.systemTools.WellformednessManager](#)),
[1725](#)
[WoodwindFingering](#) ([class in abjad.tools.instrumentTools.WoodwindFingering](#)),
[381](#)

X
[Xylophone](#) ([class in abjad.tools.instrumentTools.Xylophone](#)),
[385](#)

Y
[yield_all_combinations_of_elements\(\)](#) ([in module abjad.tools.sequencetools.yield_all_combinations_of_elements](#)),
[1107](#)

`yield_all_compositions_of_integer()` (in module `abjad.tools.mathtools.yield_all_compositions_of_integer`),
[486](#)
`yield_all_k_ary_sequences_of_length()` (in module `abjad.tools.sequencetools.yield_all_k_ary_sequences_of_length`),
[1108](#)
`yield_all_pairs_between_sequences()` (in module `abjad.tools.sequencetools.yield_all_pairs_between_sequences`),
[1108](#)
`yield_all_partitions_of_integer()` (in module `abjad.tools.mathtools.yield_all_partitions_of_integer`),
[486](#)
`yield_all_partitions_of_sequence()` (in module `abjad.tools.sequencetools.yield_all_partitions_of_sequence`),
[1108](#)
`yield_all_permutations_of_sequence()` (in module `abjad.tools.sequencetools.yield_all_permutations_of_sequence`),
[1109](#)
`yield_all_permutations_of_sequence_in_orbit()`
(in module `abjad.tools.sequencetools.yield_all_permutations_of_sequence_in_orbit`),
[1109](#)
`yield_all_restricted_growth_functions_of_length()`
(in module `abjad.tools.sequencetools.yield_all_restricted_growth_functions_of_length`),
[1109](#)
`yield_all_rotations_of_sequence()` (in module `abjad.tools.sequencetools.yield_all_rotations_of_sequence`),
[1110](#)
`yield_all_set_partitions_of_sequence()` (in module `abjad.tools.sequencetools.yield_all_set_partitions_of_sequence`),
[1110](#)
`yield_all_subsequences_of_sequence()` (in module `abjad.tools.sequencetools.yield_all_subsequences_of_sequence`),
[1110](#)
`yield_all_unordered_pairs_of_sequence()`
(in module `abjad.tools.sequencetools.yield_all_unordered_pairs_of_sequence`),
[1111](#)
`yield_nonreduced_fractions()` (in module `abjad.tools.mathtools.yield_nonreduced_fractions`),
[487](#)
`yield_outer_product_of_sequences()` (in module `abjad.tools.sequencetools.yield_outer_product_of_sequences`),
[1111](#)

Z

`zip_sequences()` (in module `abjad.tools.sequencetools.zip_sequences`),
[1112](#)